3-14-2014

REDIR: Automated Static Detection of Obfuscated Anti-Debugging Techniques

Adam J. Smith

REDIR: AUTOMATED STATIC DETECTION OF OBFUSCATED ANTI-DEBUGGING TECHNIQUES

THESIS

Adam J. Smith, Technical Sergeant, USAF

AFIT-ENG-14-M-69

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-14-M-69

REDIR: AUTOMATED STATIC DETECTION OF OBFUSCATED ANTI-DEBUGGING TECHNIQUES

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Adam J. Smith, B.S.

Technical Sergeant, USAF

March 2014

AFIT-ENG-14-M-69

REDIR: AUTOMATED STATIC DETECTION OF OBFUSCATED ANTI-DEBUGGING TECHNIQUES

Adam J. Smith, B.S.
Technical Sergeant, USAF

Approved:

//signed//	10 Mar 2014
Robert F. Mills, PhD (Chairman)	Date
//signed//	10 Mar 2014
Gilbert L. Peterson, PhD (Member)	Date
//signed//	10 Mar 2014
Michael R. Grimaila, PhD, CISM, CISSP (Member)	Date
//signed//	10 Mar 2014
Adam R. Bryant, PhD (Member)	Date

AFIT-ENG-14-M-69

Abstract

Reverse Code Engineering (RCE) to detect anti-debugging techniques in software is a very difficult task. Code obfuscation is an anti-debugging technique makes detection even more challenging. The *Rule Engine Detection by Intermediate Representation* (REDIR) system for automated static detection of obfuscated anti-debugging techniques is a prototype designed to help the RCE analyst improve performance through this tedious task. Three tenets form the REDIR foundation. First, Intermediate Representation (IR) improves the analyzability of binary programs by reducing a large instruction set down to a handful of semantically equivalent statements. Next, an Expert System (ES) rule-engine searches the IR and initiates a sensemaking process for anti-debugging technique detection. Finally, an IR analysis process confirms the presence of an anti-debug technique. The REDIR system is implemented as a debugger plug-in. Within the debugger, REDIR interacts with a program in the disassembly view. Debugger users can instantly highlight anti-debugging techniques and determine if the presence of a debugger will cause a program to take a conditional jump or fall through to the next instruction.

To my daughter. When I needed it, your precious smile always lifted my spirits. You make anything seem possible.

To my wife. More was asked of you than we ever imagined. You took it all in stride and encouraged me throughout. I could not have done this without you.

Acknowledgments

I would like to thank Dr. Robert Mills for advising me through this process. While providing unending guidance, he gave me the freedom to find my path and explore my own ability.

I would also like to thank Dr. Adam Bryant and Riverside Research (Beavercreek OH, USA). You gave me access to your software, engineers and, most importantly, expertise. This research was only possible with your support.

Finally, special thanks to Mr. Lok Yan (Air Force Research Lab, Rome NY, USA) and Mr. Edward Schwartz (Carnegie Mellon University, Pittsburgh PA, USA) for the assistance they provided using the BAP Framework.

Adam J. Smith

Table of Contents

Page

Page

Page

Page

List of Figures

List of Tables

List of Listings

List of Acronyms

Acronym	Definition
AF	Adjust Flag 7
AI	Artificial Intelligence 8
ANN	Artificial Neural Network 24
API	Application Programming Interfaces 12
AST	Abstract Syntax Tree 28
AT&T	American Telephone & Telegraph 6
BAP	Binary Analysis Platform 2
BIL	BAP Intermediate Language 27
CBR	Case-Based Reasoning 23
CF	Carry Flag 6
CFG	Control-Flow Graph 28
CPU	Central Processing Unit 5
DB	Database 32
ES	Expert System 3
ESP	Stack Pointer 13
GB	Gigabyte 43
GFRN	Generic Fuzzy Reasoning Nets 32
HMM	Hidden Markov Model 24
IA-32	Intel Architecture, 32-bit 2
IR	Intermediate Representation 2
IL	Intermediate Language 39
ITPM	Instruction Trace Pattern Matching 30
ITS	Intelligent Tutoring System 19

Acronym	Definition
MASM	Microsoft Macro Assembler 44
MinGW	Minimalist GNU for Windows 52
OEP	Original Entry Point 17
OF	Overflow Flag 6
OS	Operating System 12
OWL	Web Ontology Language 23
PE	Portable Executable 16
PEB	Process Execution Block 12
PF	Parity Flag 6
RAM	Random Access Memory 43
RCE	Reverse Code Engineering 1
REDIR	Rule Engine Detection by Intermediate Representation 2
RMI	Remote Method Invocation 42
SE	Software Engineering 33
SF	Sign Flag 6
SME	Subject Matter Expert 18
SMT	Satisfiability Modulo Theory 29
SQL	Structured Query Language 21
SS	Stack Segment 13
TF	Trap Flag 13
VM	Virtual Machine 42
ZF	Zero Flag 6

REDIR: AUTOMATED STATIC DETECTION OF OBFUSCATED ANTI-DEBUGGING TECHNIQUES

I. Introduction

1.1 Research Motivation

Reverse Code Engineering (RCE) is the process of analyzing binary programs without access to source code. Tasks such as malware analysis and software security auditing depend heavily on RCE [37]. However, RCE a time-consuming and complicated task that involves understanding computer hardware and software operations, low-level languages and logical analysis [25, 34]. RCE tools are available to help the human reverse engineer manage the complexity and facilitate the analysis process. However, anti-RCE practices can disrupt the use of RCE tools and techniques. These anti-RCE techniques complicate analysis efforts and are most prevalent in programs such as malware [45].

1.2 Problem Statement

Anti-debugging is a form of anti-RCE that attempts to prevent a debugger from properly executing the program without intervention from the engineer. With enough skill and experience, the reverse engineer can continue the RCE task with well-placed anti-debugging mitigation techniques. However, regularly during RCE, obfuscations conceal the anti-debugging techniques and make the difficult task of RCE even more challenging. A great deal of experience is required to circumvent obfuscated anti-debugging techniques efficiently. RCE analysts would benefit from a tool that could quickly identify obfuscated anti-debugging techniques for efficient mitigation.

1.3 Research Objectives

The purpose of this research is to establish the feasibility of a system for detecting obfuscated anti-debugging techniques in programs. To achieve this goal, a debugger plug-in was developed to statically analyze binary programs to detect these techniques. Debugger users will launch the detection process from within the debugger and view the implemented technique in the disassembly view. The lines of code that comprise the technique will be highlighted for the user. The process for detecting these obfuscated anti-debugging techniques will follow a sensemaking theory for information gathering and understanding.

1.4 Approach

The Rule Engine Detection by Intermediate Representation (REDIR) system for automated static detection of obfuscated anti-debugging techniques is a prototype designed help the RCE analyst quickly, and correctly, avoid anti-debugging techniques.

Sensemaking offers a theory that allows for the development of minimal information into a complete information gathering task [51]. Employing a sensemaking strategy, the initial technique detections from a rule-based system is developed through a process that will discard false detections and promote possible detection candidates. To develop these candidates, an Intermediate Representation (IR) tool translates programs into the IR's simplified language, create small sub-programs to encapsulate the technique, and conduct evaluations to determine if the addition of simulated debugging conditions will cause the chop to terminate with a different outcome.

REDIR relies on several principles to afford identification of obfuscated, anti-debugging techniques. Based on the *Data/Frame* sensemaking process, REDIR develops minimal starting information into a confirmed detection [51]. First, the Binary Analysis Platform (BAP) Framework translates the program into an IR that converts the Intel Architecture, 32-bit (IA-32) instruction set down to a much smaller set of semantically

equivalent statements [23]. Next, the IR is parsed into an Expert System (ES) rule-engine to search the IR for instances of any anti-debugging technique characteristics. These minimal characteristics are IR statements that represent the beginning and ending of a technique based on minimal and unavoidable heuristics. The IR results from the rule-engine form the bounds for chopping the program down to a small sub-program of IR statements. An instrumentation process then adds code to simulate a debugging condition in the IR chop. Finally, concrete evaluation by taint analysis of the instrumented IR chop reveals the presence of the technique.

Unlike other solutions to detect these techniques, REDIR is static. Similar methods for detection use dynamic instruction traces to determine the presence of the desired code feature [46, 75]. Additionally, these methods report detections but do not offer them back for analysis. Conversely, REDIR will highlight the detected anti-debugging technique in the debugger disassembly view.

1.5 Research Limitations

Testing a tool for detecting anti-debugging techniques involves one of two test strategies. The first strategy involves using real-world programs such as malware to evaluate if it can detect the anti-debugging technique. The problem with this strategy is that there are no existing test data sets that are statically analyzable and guaranteed to implement the anti-debugging techniques under test. The second strategy involves creating test data. The problem with this strategy is that tests conducted on these programs are not guaranteed to work on real programs.

For this research, a test corpus was created to combine known anti-debugging techniques with common obfuscations. Two key considerations drove this decision. First, the techniques implemented in the test corpus derive from documented implementations found in real-world programs. Second, even if these specific implementation combinations do not exist in real-world malware, this research did not intended to search for

specific implementations. The goal of this research is to find the anti-debugging technique, regardless of implementation. Finding the various anti-debugging technique implementations contained in the test corpus attests to the feasibility of the system and the potential ability to find other unknown implementations.

1.6 Thesis Overview

This thesis proposes a method for static detection of obfuscated anti-debugging techniques based on a rule engine sensemaking process with the aid of IR. Chapter 2 reviews the concepts and technologies that contribute to this research ranging from an exploration of RCE concepts, through ES technologies, sensemaking and finally on to related work. Chapter 3 describes the system design, implementation limitations and testing methodology. Chapter 4 details the results of the REDIR analysis of the test corpus including detailed analysis of six representative test cases. Lastly, Chapter 5 summarizes this document and provides new research avenues based on this work.

II. Literature Review

2.1 Introduction

The following sections provide the necessary background for the remainder of this thesis. First, Section 2.2 provides an overview of Reverse Code Engineering (RCE) knowledge, tools and techniques. Next, Section 2.3 presents a breakdown for the various Expert System (ES) technologies. Then, Section 2.4 introduces the concept of Intermediate Representation (IR) and the Binary Analysis Platform (BAP) framework. Finally, Section 2.5 overviews similarities and differences of related works.

2.2 Overview of Reverse Code Engineering

RCE is the process of extracting details about a software program from the binary executable. A number of motivations drive RCE such as reengineering of a system as a whole, malware detection and analysis, and the "cracking" of copy protection. In order to accomplish this work, engineers depend on resources that fall into three general categories: knowledge, interface, and task. The following subsections provide a brief overview of each.

2.2.1 Knowledge

2.2.1.1 Assembly Language

Assembly language is the lowest-level programming language designed for human comprehension [47]. Assembly is actually a set of mnemonics that describe machine instructions processed by the Central Processing Unit (CPU). The ability to mentally process disassembled application code is the root of the RCE process. Unfortunately for reverse engineers, the assembly language outputs provided by disassemblers or debuggers are not nearly as easy to comprehend as original source code [70]. These outputs do not provide code comments or descriptive function names [47]. Assembly code does not provide any details about higher-level data structures or variable types. Assembly

instructions represent the low-level work of the processor. Data values move from memory to registers and back. The processor manipulates *pointers* to control program flow and store data. Tools can convert these operators represented in the executable by binary zeroes and ones into a slightly more readable format. The availability and functionality of operations depends on the processor the code was compiled for. Additionally, the reverse engineer can choose their preferred syntax for assembly code; x86 has Intel and American Telephone & Telegraph (AT&T) syntaxes. For these reasons, assembly code is notoriously difficult to read and comprehend [70].

Listing 2.1 and Figure 2.1 provide two different representations of a simple Hello World C program. Listing 2.1 shows a simple Hello World C program written in C. The exact same program in Figure 2.1 is shown disassembled in IDA Pro. This representation is much longer and far more unreadable than Listing 2.1.

```c
#include<stdio.h>
main()
{
  printf("Hello World!\n");
}
```

Listing 2.1: helloworld.c: Hello World C program.

Furthermore, each assembly instruction can have many side effects [23]. For example, A simple `XOR` instruction, designed to implement an *exclusive or* ($\oplus$) operation, has several. Commonly, the instruction `XOR EAX, EAX` is used to zero-out the value of a register since, for any value x, $x \oplus x = 0$. In addition to the change of `EAX`, several control flow flags are also changed. According to the Intel Architecture, 32-bit (IA-32) language documentation, the `XOR` instruction can cause the "Overflow Flag (OF) and Carry Flag (CF) flags are cleared; the Sign Flag (SF), Zero Flag (ZF), and Parity Flag (PF) flags are set according to the result.

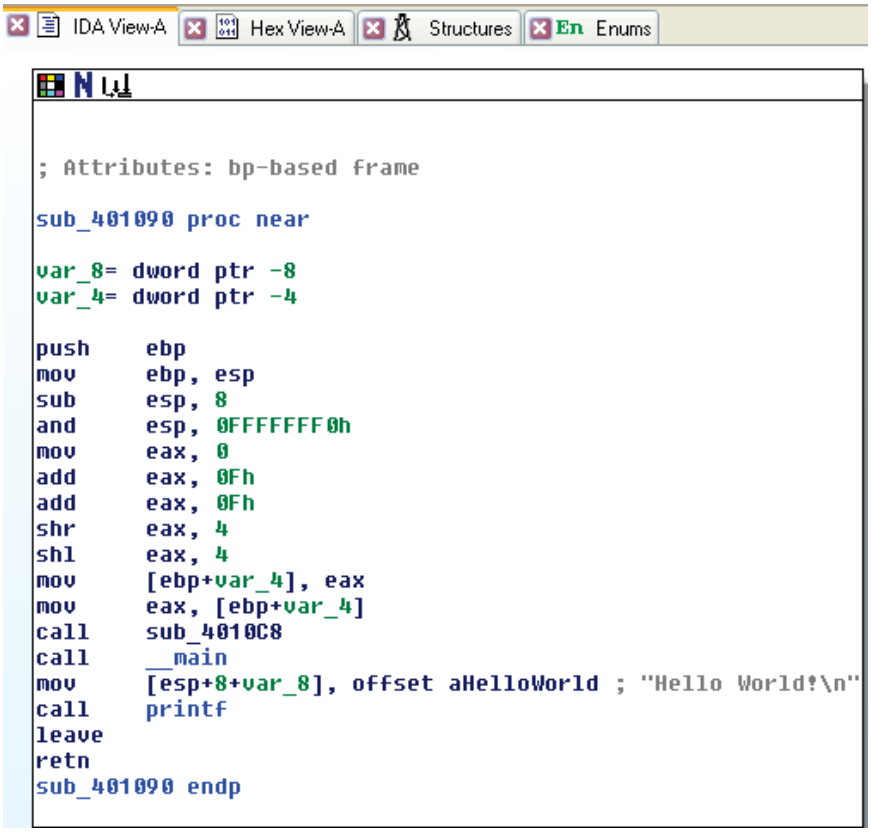

Figure 2.1: Portion of the compiled helloworld.c program in IDA Pro Dissembler.

The state of the Adjust Flag (AF) is undefined" [7]. The reverse engineer must be aware of the explicit assignment operation but also any side-effects.

2.2.1.2 Explicit vs. Tacit Knowledge

Reverse engineering software requires the possession of a variety of skills and knowledge. Some of these skills are precise and explicit such as the assembly language syntax [49]. Explicit knowledge forms the foundation of the skill and can be acquired in a classroom or from a book. Unfortunately, limited classroom opportunities are available for such learning. However, automated tools can tackle problems based on explicit knowledge to help the reverse engineer.

Research suggests that some level of expertise in an activity can be developed in as little as 50 hours of deliberate practice while world-class skill requires as much as 10,000 hours [36]. A reverse engineer's tacit knowledge base grows as he or she progresses from 50 to 10,000 hours. Unlike explicit knowledge, tacit knowledge develops through practice. Reverse engineers develop "rules of thumb" that guide them through the process [49].

Knowledge for cognitive support has not been formalized for the software engineering discipline [73]. Individuals have created tools to aid cognition but only by the creators' hands-on intuition rather than scientific principle. Tacit knowledge disciplines, like RCE, fall into the category of *craft discipline*. As disciplines progress, they transform their tacit knowledge base into explicit knowledge. Unfortunately for the field of RCE, several issues have slowed that progression [63].

2.2.1.3 Reverse Code Engineering as a Sensemaking Task

Psychologists describe sensemaking as a composite process that incorporates creativity, curiosity, comprehension, mental modeling and situational awareness [50]. Klein, et al. describe sensemaking as a "motivated, continuous effort to understand connections...in order to anticipate their trajectories and act effectively". To be an effective aid to sensemaking, a joint human-Artificial Intelligence (AI) team must be "mutually predictable", "directable" and share a "common ground" (understanding) of the domain and problem [52].

Bryant, et al. identified the process of analyzing a program executable as an example of a sensemaking process that occurs between a human and a system [25]. To be effective, a joint human-AI RCE team must be able to accomplish several key goals. First, they must establish predictability by agreeing on a specific RCE workflow. Next, they must give and take direction by either learning or applying learned RCE knowledge. Finally, the team must share a common view of the problem. This view should resemble an artifact of the RCE process [70].

Klein, et al. introduce the Data/Frame sensemaking theory as a process where understanding develops through a transition between *frames* [51]. These frames form into a closed loop encompassing the life of the sensemaking task (see Figure: 2.2). Frames are created based on only minimal data. Through a process of *questioning*, *elaborating*, and *reframing*, the frame is refined for the life of the task.

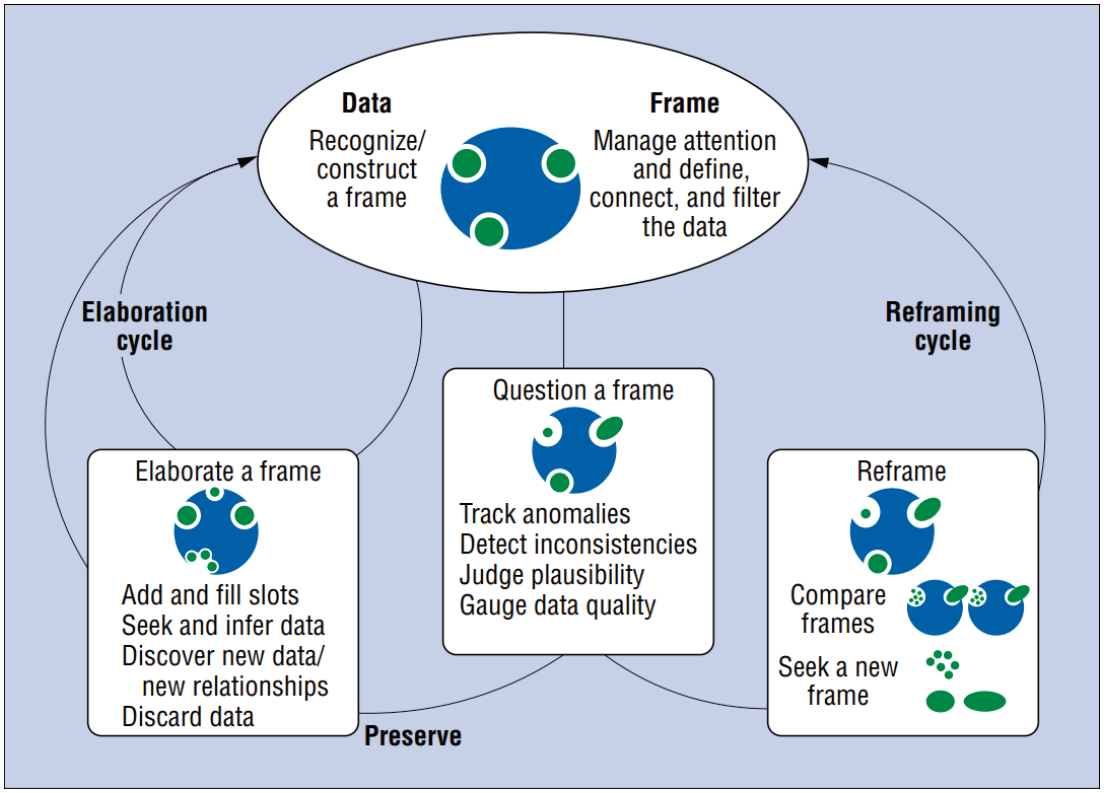

Figure 2.2: Klein, et al. Data/Frame sensemaking theory [51].

2.2.2 Interface

2.2.2.1 Compilers

Compilers are responsible for converting higher-level code written by programmers into the executable files of machine instructions [13]. Compiler design strives to achieve the best possible performance and have considerable influence over how a program converts

into machine instructions. The optimizer works to reduce the processor cycles required for a series of instructions by reordering and replacing the instructions. Compilers will remove redundant calculations, move code out of loops, and reorder instructions as necessary. Additionally, there are numerous, processor dependent, implementations of these optimization schemes. Often these optimizations result in assembly code that is non-intuitive and difficult for the human reverse engineer to understand.

2.2.2.2 Common Reverse Code Engineering Tools

2.2.2.2.1 System Monitors As programs execute they act upon the system affecting memory use, network access, hard disk access, and other functions or resources [34]. System monitors can observe the interaction between the program and the operating system. Reverse engineers can use system monitors to analyze the behavior of the program they are analyzing to determine what the program does or if it acts maliciously.

2.2.2.2.2 Disassemblers Disassemblers are the most basic and essential reverse engineer tools [34]. Disassemblers work by transforming the binary instructions of a program into their assembly language representations. Disassemblers are processor specific; however more capable disassemblers support a variety of processor architectures.

2.2.2.2.3 Debuggers Debuggers step through an application, line by line, to give the user a view of a program while it is executing [34]. Reverse engineers can use debuggers to pause code execution with breakpoints and trace instructions in a program. Debuggers include simple disassemblers to allow the reverse engineer to watch the code as it is processed by the CPU.

2.2.2.2.4 Decompilers A Decompiler can convert the executable back into a higher-level language that resembles the original source code [34]. In most circumstances, decompilers cannot reverse the entire program. With their limitations,

decompilers still find use in RCE. Even if the decompilation attempt is partly successful, the result can provide information that will save the reverse engineer time and effort.

2.2.3 Task

2.2.3.1 Anti-Reverse Engineering Software Design

Software design plays a major role in the effort required of reverse engineers. The designers of proprietary software and malware employ tactics to disrupt the reverse engineering of their code [67]. Simple but effective techniques include using encryption and compression to occupy the reverse engineer's time and effort. More sophisticated techniques like those listed below make the engineer's work quite difficult.

2.2.3.2 Encryption and Compression

To avoid detection, malware writers can encrypt or compress data portions of the code as a form of obfuscation [45]. Searching data sections for clues becomes difficult as a result. To learn the function of the software, reverse engineers must manually de-obfuscate each data area. This manual process can become quite time consuming and tedious. Automating the de-obfuscation is possible with debuggers that can step through the code, decrypting or decompressing as they go. Unfortunately, malware designers can use anti-debugging techniques in conjunction with encryption and compression to counter the use of debuggers.

2.2.3.3 Anti-debugging

Unlike disassembly which analyzes static executables, debuggers look at the code as it is executing. Unfortunately for reverse engineers, debuggers can be fooled with simple tricks [67]. Calls to *system interrupts* can force the debugger to lose context while analyzing. To detect debuggers, the program can generate checksums for portions of code as they exist in the execution stack. The breakpoints inserted by the debugger add to the checksum calculation and the mismatch becomes easily detectable. Debuggers also often save a trace record to the stack. Analyzing the stack at certain points in execution can reveal part of this trace to the program. Simple anti-debugging techniques include

using specific Application Programming Interfaces (API), checking the debugger's registry values, searching memory for specific debugger strings (e.g. "Ollydbg"), or scanning for the particular drivers used by debuggers. The following techniques are debugger-agnostic examples of anti-debugging possible in user-level code.

2.2.3.3.1 Operating System Flags The easiest method to detect if a debugger is in use is to look for specific flags set by the Operating System (OS) [45]. These flags are normally made available through invocation of the 32-Bit Windows API calls `isDebuggerPresent()` or `isRemoteDebuggerPresent()`. Additionally, these OS flags can be checked manually. The code in Listing 2.2 checks for a debugger by looking at the Process Execution Block (PEB) for the byte used by `isDebuggerPresent()`.

```
1 mov eax, fs:[30h]
2 movzx eax, byte ptr [eax+2]
3 test eax,eax
4 jne DebuggerFound
```

Listing 2.2: IA-32 implementation example of manually testing the PEB isDebuggerPresent byte.

2.2.3.3.2 Timing Another method for detecting the presence of a debugger is to use a timing comparison. By checking the time twice, once before and again after code segment, the program can detect if its execution was delayed. The code in Listing 2.3 demonstrates a timing-based detection technique using the IA-32 `rdtcs` instruction which returns the number of processor cycles since startup.

```
1 rdtsc
2 xor ecx, ecx
3 add ecx, eax
4 rdtsc
5 sub eax, ecx
6 mov ecx, 0FFFh
7 cmp eax, ecx
8 ja DebuggerFound
```

Listing 2.3: IA-32 implementation example of RDTSC Timing detection technique.

2.2.3.3.3 Interrupt Handling

Interrupt handling techniques are very effective since they prey upon the debugger's handling of the interrupt and the user's incomplete understanding of the underlying operations [38, 45]. These techniques attempt to have the debugger change the data stored in flags and registers or act inappropriately. The "move stack segment" (MOV SS) technique is interesting because when a value is set to the Stack Segment (SS) register, the CPU will covertly set the Trap Flag (TF) in a special, multi-purpose data structure known as the *EFLAGS* register. Next, while debugging, the CPU will advance the Stack Pointer (ESP) and the debugger will clear the flag [34, 38]. While single stepping over the instruction, the debugger will seem to skip to the next instruction. This is because the TF will disable the debugger's next single step interrupt. If that next instruction happens to persist the TF by pushing it onto the stack, the value is preserved and used later to direct control flow. Testing the TF will inform the program that the debugger cleared the TF. In Listing 2.4 the `pop SS` instruction covertly sets the TF. The `pushfd` instruction then pushes the *EFLAGS* image onto the stack. Now, the TF is available at any time for use in a control-flow decision.

```
1 push ss
2 pop ss
3 pushfd
4 test word ptr [esp+1], 1
5 jne DebuggerFound
```

Listing 2.4: IA-32 implementation example of the MOV SS detection technique.

2.2.3.4 Obfuscation

If malware authors all wrote malware the same way, the job of analyzing malware would be quite easy. To make analyzing programs more difficult, obfuscation techniques can disguise the true nature of a program.

2.2.3.4.1 Layout Obfuscation

Layout obfuscation techniques attempt to confuse the analyst by concealing important instructions among other irrelevant instruc-

tions [56]. Simple techniques include insertion of dead code (nop and other non-functional instructions) between the functional instructions [76]. Reassigning registers between code segments can further disrupt the analyst. The example shown in Listing 2.5 is the same program from Listing 2.2 with the instructions reordered in a process called *code transposition*. The use of labels makes this example easy to comprehend. When displayed in a debugger (see Figure 2.3) this code is more difficult to read.

```
jmp step1
step3:
  test eax,eax
  jmp step4
step2:
  movzx eax, byte ptr [eax+2]
  jmp step3
step4:
  jne DebuggerFound
  jmp end
step1:
  mov eax, fs:[30h]
  jmp step2
```

Listing 2.5: IA-32 implementation example of manually testing the PEB isDebuggerPresent byte with code transposition applied.

```
00401000  .text          Entry Point    eb 18              jmp short _start
00401002  .text          loc_401002     85 c0            =>test eax, eax
00401004  .text                         eb 06              jmp short loc_40100C
00401006  .text          loc_401006     0f b6 40 02      =>movzx eax, byte ptr [eax+2]
0040100A  .text                         eb f6              jmp short loc_401002
0040100C  .text          loc_40100C     75 14            =>jnz short loc_401022
0040100E  .text                         68 0f 30 40 00     push string_40300F          "No Debugger Found"
00401013  .text                         e8 20 00 00 00     call sub_401038
00401018  .text                         eb 14              jmp short loc_40102E
0040101A  .text:_start   _start         64 a1 30 00 00 00  mov eax, large fs:30h
00401020  .text:_start                  eb e4              jmp short loc_401006
00401022  .text          loc_401022     68 00 30 40 00   =>push string_403000          "Debugger Found"
00401027  .text                         e8 0c 00 00 00     call sub_401038
0040102C  .text                         eb 00              jmp short loc_40102E
0040102E  .text          loc_40102E     6a 00            =>push 0
00401030  .text                         e8 ad 00 00 00     call sub_ExitProcess
```

Figure 2.3: Program from Listing 2.5 as displayed in DigR Debugger.

2.2.3.4.2 Conditional Code Obfuscation

Conditional code obfuscation techniques hide the intended execution paths of programs [66]. The strength of these

techniques is that static analysis becomes very difficult as no one true execution path is detectable; dummy code presents a valid execution path.

One such method of conditional code obfuscation is an opaque predicate [29]. Here, the predicate (cause for some control flow decision) is unknown. The opaque predicate is expressible in terms of predicate P and program p. The predicate can evaluate always true P_p^T, always false P_p^F, or neither $P_p^?$ if it does not always point the same direction.

For example, observe Listing 2.6. This is an example of a P_p^T opaque predicate. This program employs the algebraic identity $(x+y)^2 = x^2 + 2xy + y^2$ to form a *number-theoretic* opaque predicate which always evaluates true [18]. As a result, 26 lines of code have disguised a single unconditional jump. A human reverse engineer would require additional time to analyze this jump and a static analysis tool would likely be unable to determine the correct jump direction, especially if x and y were runtime variables. It is also important to note that applying other obfuscations to an opaque predicate will make analysis more difficult. A human reverse engineer would not likely see the entire algorithm laid out neatly for analysis.

```
xor eax, eax
add ax, x
add ax, y
imul ax, ax
push ax      ; push (x+y)^2
xor eax, eax
mov ax, x
imul ax, ax
push ax      ; push x^2
xor eax, eax
mov ax, y
imul ax, ax
push ax      ; push y^2
xor eax, eax
xor ebx, ebx
mov ax, x
mov bx, y
imul ax, bx
imul ax, 2   ; ax = 2xy
pop bx       ; bx = y^2
add ax, bx   ; ax = 2xy + y^2
pop bx       ; bx = x^2
add ax, bx   ; ax = x^2 + 2xy + y^2
pop bx       ; bx = (x+y)^2
cmp ax, bx   ; always evaluates true: ax == bx
jne fake     ; never jumps
```

Listing 2.6: IA-32 implementation example of a P_p^T *number-theoretic* opaque predicate.

2.2.3.5 Anti-heuristics

Heuristic analysis is a tool used by reverse engineers to detect viruses based on their similarity to other known viruses [67]. Many viruses use "packers" to package their virus together with a harmless executable to conceal its presence Figure 2.4 depicts an executable program packed within a Windows Portable Executable (PE) file [62, 69]. While packed, the concealed portion of the code is encrypted, encoded, or obfuscated to hide its implementation. The packer works by revealing the concealed code as necessary to execute it. Additionally, malware packers employ the other anti-reverse engineering tactics mentioned previously to make unpacking difficult. Malware analysts must carefully unpack the executable by defeating the anti-reverse engineering techniques to discover the

first instruction for the program known as the Original Entry Point (OEP) of the unpacked malware.

Typically, once the OEP of the malware has been identified, the program is dumped from memory into an unpacked executable for further analysis. To hide from heuristic analysis, virus writers now use sophisticated packers that can embed malware deep within virtually any file format. Nesting viruses deep within a tree of executables, compressed archives and data files complicates heuristic methods. Furthermore, virus writers will pack multiple executable sections (including multiple viruses) into an executable to hide the true entry point of the program from the analyzer.

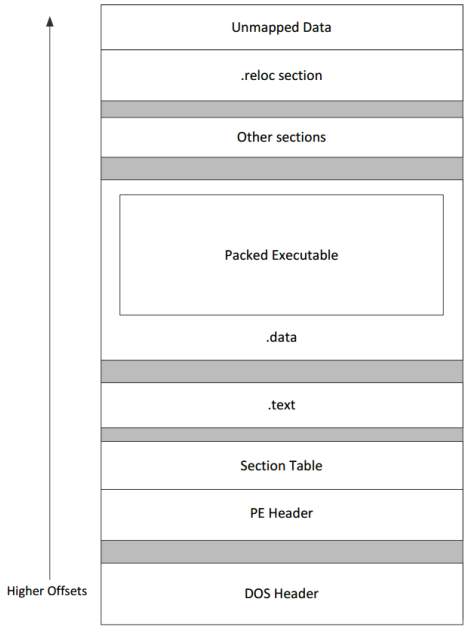

Figure 2.4: Portable Executable (PE) depicted with embedded packed executable [62, 69].

2.2.3.6 Mutation

As anti-virus technology has progressed, the work of hiding malware has become more difficult. As a result, malware developers have developed mutation techniques to avoid detection. Encrypted viruses use variable decryption schemes to insert dummy

instructions in the code [67]. These instructions make a previously known sample appear new. Oligomorphic techniques increase the complexity of encrypted viruses by adding additional decryptors to the virus. An oligomorphic virus would randomly select from the available decryptors at runtime adding to the possible variations. Polymorphic viruses include mutating decryptors that are capable of creating millions of unique virus samples. Metamorphic viruses on the other hand swap modules around within the executable creating new forms. Metamorphic viruses have the ability to create $n!$ permutations (for n subroutines). A simple metamorphic virus, *Badboy*, contained eight subroutines for $8! = 40,320$ permutations.

2.3 Expert Systems Overview

Research in the field of AI has spanned many decades and created numerous foundational technologies [55]. ES have become a cornerstone of AI research and implementation. The concept behind an ES is simple: transfer knowledge of a particular domain from a Subject Matter Expert (SME) to a computer system. Once in the system, this knowledge can find a variety of uses. The typical ES is composed of three parts: a knowledge base, a working memory and an inference engine [31]. The knowledge base is simply the storage place for the expert knowledge. The working memory stores the details of the current problem including the user input and program inferences. The inference engine performs the work of leveraging the knowledge base against the working memory to learn new information about the situation.

Many variations of the original ES exist. For the remainder of this paper, more specific variations are discussed and evaluated instead of the generic ES. Furthermore, while these technologies are unique, they are not mutually exclusive. Many applications are hybrids of ES technologies. Figure 2.5 provides an illustrative overview of ES.

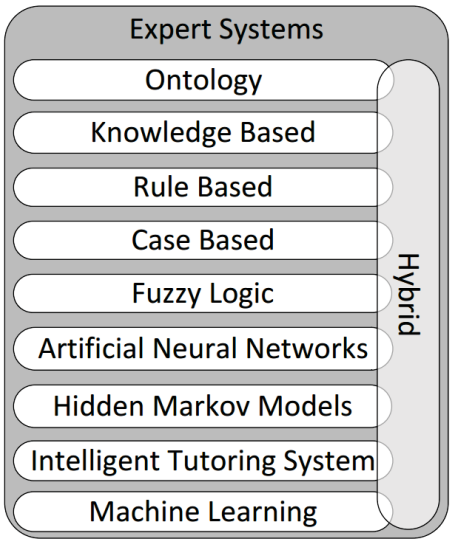

Figure 2.5: ES overview.

2.3.1 Intelligent Tutoring Systems

An Intelligent Tutoring System (ITS) is designed to use AI to provide an education or tutoring experience for a student [60]. ITS can be characterized by an ES knowledge module like a rule-based or knowledge-based system packaged with a student model, a tutoring module and a user interface. The student module maintains a representation of the student's understanding based on his or her progress through the lesson. This student information is fed to the tutoring module as the basis for specific tutoring decisions. The tutoring module works by dividing lessons into a series of *tasks* and *steps* (see Figure 2) [72]. An outer loop selects tasks by one of four methods: student selected, fixed progression, mastery of lesson knowledge or *macroadaptive* (adapted based on student performance). The sequence of steps in the inner loop derives from the student and tutoring module's determination for the most effective learning. The inner loop is where the tutoring module determines the amount of feedback and assistance to provide. When called, the *Step Generator* returns the next step for the student to perform. Other interpretations of

ITS systems contain additional modules such as the domain expert model that represents the ideal solution model and the bug catalog which lists common domain errors and misconceptions [30]. Figure 2.6 provides an illustration of an ITS.

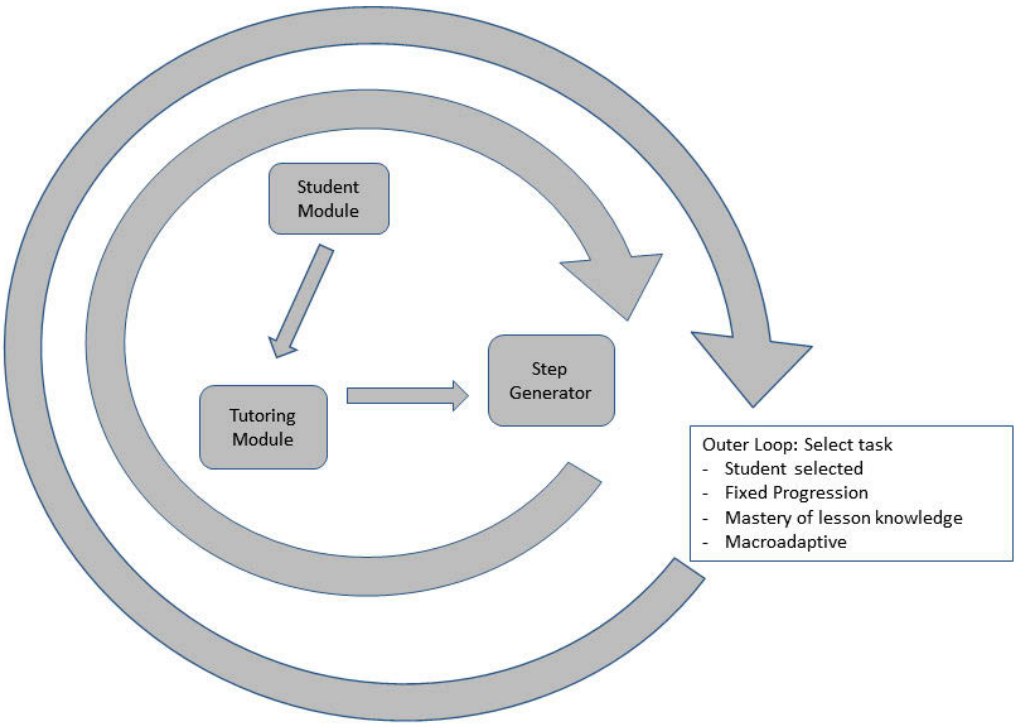

Figure 2.6: Illustration of an ITS [72].

ITS systems have already proved useful in instructional settings [61]. They are well suited for basic skill instruction where the student knows how to ask the appropriate questions. Domains with well-defined problem spaces have the most positive impact from ITS. These systems still have only had limited success due to sensory limitations. ITS systems have only begun to factor in other measurements of the student beyond keyboard inputs such as eye movement and vocalization monitoring. Human tutors can quickly detect a student's lack of interest, frustration or boredom [22]. ITS systems will have limited impact until the emotional state of the student adequately incorporates into the system [61].

ITS systems have been used in several computer science domains [55]. The lessons learned from creating tutoring systems for teaching the LISP programming language, enhancing cognition and teaching Structured Query Language (SQL) programming could inspire future RCE ITS [16, 17, 58].

2.3.2 Rule-Based Expert Systems

Rule-based ES are designed to codify knowledge provided by SMEs based on an easy to understand syntax [55]. This knowledge forms rules that are expressible in *if-then* syntax. Rule-based systems are composed of three parts: the working memory, rule base and the inference engine [65]. The working memory maintains the current state of situation based on a known set of facts. The rule base (knowledge base) provides the complete knowledge of the domain provided by the SME. The inference engine uses the working memory and the rule base to learn new information about the problem.

Rule-based systems have several advantages over other AI systems. Rule-bases have a uniform syntax such as `ruleid: If antecedent1 and antecedent2 .... then consequent`. This syntax makes the rules self-documenting and easy to understand. Rules are also independent since each rule represents one fact about a particular domain. Additionally, rules can be prioritized to optimize processing. Finally, rules are useful as computational models based on their programmatic syntax.

There are several disadvantages to rule-based ES as well. All rules exist on the same level; they cannot be represented in a hierarchy. As a result, all rules cycle through before selecting the appropriate rule. Rule-based systems also become tedious when representing human problem solving as a single task breaks down into numerous atomic subtasks.

Rule-based systems have been used successfully in numerous domains [55]. Most notably relating to RCE, rule-based systems have been used in teaching, knowledge base maintenance, knowledge acquisition, knowledge representation and tutoring systems [28, 41, 42, 44, 74].

2.3.3 Knowledge-Based Expert Systems

Similar to rule-based systems, knowledge-based ES can replicate limited human knowledge provided by SME into a computer systems [55]. Typical knowledge-based systems are characterized by four components: a knowledge base, an inference engine, a knowledge engineering tool and a user interface. Unlike rule-based systems, knowledge-based systems do not have a prescribed syntax. The purpose of the knowledge engineering tool is to add knowledge to the system. This process can either be human directed or automated [14]. The user interface in a knowledge-based system should provide, as natural as possible, access to the knowledge stored in the system.

There are several advantages and disadvantages to using knowledge-based systems [20]. Advantages include the ability to make mostly inaccessible information widely available. Additionally, the knowledge base serves to preserve the knowledge for the future. Unfortunately, if the knowledge base has errors or is incorrect, that incorrect knowledge is also preserved. In addition, the knowledge base does not contain the common sense or intuition of the SME.

Knowledge-based AI approaches have been employed for decades. Many of those applications have aspects that relate to creating a tool for RCE. Knowledge-based tools have been applied in knowledge management, knowledge representation, decision -making and learning [32, 57, 59].

2.3.4 Ontologies

Ontologies are vocabularies that provide a common communication domain model between SMEs and software developers [55]. These vocabularies can be structured in many ways from "highly informal," like natural speech, to "rigorously formal" with rigid semantics [71]. Ontologies are useful because they formalize domain knowledge creating a shared understanding of a problem. Additionally, ontologies allow knowledge reuse. Once the ontology is built, the knowledge base can port to tools or other problem

domains. Unfortunately, ontology creation cannot be automated. The current process is manual, time-consuming, and requires cooperation between domain experts and ontology developers.

Use of ontologies is common in a number of domains. Several examples exist that could have implications on RCE. Knowledge reuse, knowledge acquisition and knowledge modeling activities have all employed ontologies [39, 64, 68].

Ontologies are very useful tools for codifying expert knowledge. They can use strict or fuzzy semantics to identify dynamic context and conditions. In addition, ontologies append easily to software systems using existing semantic reasoner software libraries such as Pellet and Hermit [6, 10]. Additionally, the Web Ontology Language (OWL) provides a standard, implementation agnostic format for specifying ontologies [9]. OWL ontologies work with several software projects using different semantic reasoner libraries.

2.3.5 Case-Based Reasoning

Case-Based Reasoning (CBR) is unique among the other AI problem solving strategies [12]. Instead of employing knowledge learned from SMEs, CBR learns by remembering previous solution cases and applying best match solutions to new cases. Solutions that pass verification become learned cases and add to the knowledge base for future use. Failed solutions are remembered as well as incorrect solutions for future use. The benefit of CBR is that it emulates one of the way humans solve new situations. The downside of CBR becomes evident when the system accepts an incorrect case as learned. The incorrect case must be removed from the knowledge base for the system to function properly.

In 2010, Gutierrez-Santos, et al. employed CBR in an ITS to create an exploratory learning environment [40]. The environment allowed the students to create free-form models and structures based on observed properties and relationships. A similar approach would be useful in RCE in an assembly feature recognition module.

2.3.6 Artificial Neural Networks

Artificial Neural Networks (ANNs) take AI a bit farther than other strategies by attempting to recreate a portion of the actual biology of the intelligent being they are trying emulate [55]. Typical applications of ANNs focus on achieving performance of highly parallel processing similar to that of biological organisms. Networks of artificial neurons characterize ANN designs. Each neuron produces an individual output based on the signals received from the rest of the network [43]. Meta-heuristic algorithms can optimize the application by training the network based on particular parameter values. The strength of ANNs is their ability to solve complex, nonlinear classification problems despite the simplicity of individual neurons. Conversely, ANNs can be slow to train and can suffer from over training.

2.3.7 Hidden Markov Model

Hidden Markov Models (HMMs) are statistical models that serve to analyze complex random situations [43]. HMMs apply to time series or linear sequences to reveal important unidentified states [33]. HMMs are effective at identifying a condition based on partial sequences of states [43]. Additionally, they can function as prediction algorithms due to their ability to function with a partial sample of observations.

Cha, et al. proposed a HMM-based ITS [27]. In experiments against a decision tree method of detecting learner style, the HMM approach led to an error rate half that of the decision tree. Without proper training sequences, HMMs can identify and predict incorrect situations. Finally, HMMs are computationally expensive compared to other systems.

2.3.8 Machine Learning

Machine learning is a significant branch of AI that focuses on creating systems with the ability to learn [15]. Learning can be implemented in many different ways by employing techniques including ANNs and HMMs. Additionally, machine learning can be *supervised* or *unsupervised*. Supervised systems rely on a human supervisor to provide correct samples

for analysis. Machine learning systems perform well at pattern matching and knowledge extraction tasks. The primary disadvantage of using machine learning in an expert system is accuracy. "Hand-crafted" ES knowledge bases achieve greater accuracy than ESs with knowledge bases created by machine learning [21].

2.3.9 Fuzzy Logic

Not all problems fit neatly into a particular state. Fuzzy logic relaxes matching criteria to allow applications to be less precise in order to deal with uncertainty the way a human might [26]. Systems that use fuzzy logic often attempt to work within a set of bounds rather than an exact value. These systems are able to make decisions based on subjective analysis. During development of fuzzy logic based systems, designer still must account for all possible states. Fuzzy logic can detect if a set of measurements exists within a particular state; it cannot detect new, unknown states.

Fuzzy logic has been used successfully in many different AI systems. An implementation of a traffic light control system demonstrated how fuzzy logic could improve traffic flow based on traffic density measurement. Lexicographical search algorithms can use ontologies of synonyms to improve search results [54].

2.3.10 Summary

ES are useful because they allow SMEs to codify their knowledge in an information system. The purpose of this section was to describe the advantages, disadvantages and possible uses of the available ES technologies. Table 2.1 on page 26 provides a concise summary of this section.

Table 2.1: Summary of advantages and disadvantages for ES technologies

ES Technology	Advantage(s)	Disadvantage(s)
ITS	Well-suited for basic instruction [61]	Cannot detect frustration, boredom, or loss of interest [22]
Rule-Based	Self-documenting, independent, and useful as programming models [65]	Rule structure flat (non-hierarchical), all rules must be evaluated for each check, and tedious to program [65]
Knowledge-Based	Increases availability of expert knowledge and preserves for future use [20]	Perpetuates incorrect knowledge and lacks common sense and intuition of expert [20]
Ontologies	Formalize domain knowledge and provide for knowledge reuse [71]. Standardized format, easy to use in software systems [9]	Time consuming to generate and requires collaboration between domain experts and ontology developer [71]
Fuzzy Logic	Allows for less precise condition matching [26]	Designers must still account for all possible condition states [26]
CBR	Emulates human learning [12]	Incorrectly learned information must be purged from the set of cases [12]
ANN	Capable of complex problem solving and individual neurons are simple [43]	Slow to train and can be overtrained [43]
HMM	Effective at identifying conditions based on partial sequences of states and useful as prediction algorithms [43]	Can identify incorrect states without proper training and are computationally expensive [27]
Machine Learning	Well-suited for pattern matching and knowledge extraction tasks [15]	Less accurate than "hand-crafted" ESs [21]

2.4 Intermediate Representation and Binary Analysis Platform Introduction

2.4.1 Intermediate Representation of Assembly Language Programs

Excluding floating-point and other special purpose instructions, the Intel 64 and IA-32 instruction set contain 254 unique general-purpose instructions [7]. Based on the complexity of the instruction set and the presence of side-effects, assembly-only analysis becomes very difficult [19]. Use of an IR will expose hidden operations and form an abstraction for a robust instruction set architecture [53]. Several IR implementations exist for disassembling and translating x86 programs. The following paraphrases the requirements established by Koschke, et al. for an effective IR [53].

R1 Programming language independent

R2 Well-defined semantics that exactly describe the constructs used in the modelled programming languages

R3 IR traversals should be efficient

R4 IR construction should be efficient

R5 Length of IR should be linear to modeled source code

R6 IR should permit control and data flow analysis efficiently

R7 IR should map to original source code

R8 IR should be able to describe a system composed of several programs

R9 Support various granularity levels based on use-case

R10 IR should retain user all comments and other information beyond the original source code

R11 The IR should be saveable

R12 Adding code construct abstractions to the IR will not invalidate previous analysis

R13 IR should represent higher-level concepts

R14 IR should permit multiple views in multi-user environments

2.4.2 Binary Analysis Platform

The BAP is a framework of tools designed to create and manipulate IR of executable programs [24]. BAP is an ongoing project at Carnegie Mellon University and has an active support community.

2.4.2.1 Semantics

The BAP Intermediate Language (BIL) is the IR form used by the BAP framework of utilities. BIL will decompose individual disassembled instructions into one or more statements. There are just seven different types of statement in BIL (*var := exp*, *jmp*, *cjmp*,

halt, *assert*, *label* and *special*) and all have zero side-effects. These statements reduce massive instruction sets down to simple intuitive operations.

Several BIL expressions go into an individual BIL statement. BIL statements use the following expressions to describe instructions. The *load* expression describes any activity where reading memory and storing the contents in another location. The *store* expression is the inverse of *load* as it describes when writing to memory. Additionally, expressions can take the form of binary and unary operations. The remaining expressions (*lab*, *cast*, *let*, *unknown*, and *name*) represent less frequent operations.

2.4.2.2 Utilities

2.4.2.2.1 toil The primary purpose of the *toil* utility is to convert executable programs into BAP BIL. Programs analyzed with *toil* are first "lifted" to the BIL. Listing 2.7 demonstrates IA-32 instruction `XOR EAX, EAX` lifted to BIL. In this example, `R_EAX:u32` represents the destination register. The remaining BIL statements expose the side effects of the XOR instruction. Additionally, the *toil* utility can lift dynamic traces into BIL.

```
addr 0x40100e @asm "xor    %eax,%eax"
label pc_0x40100e
R_EAX:u32 = 0:u32
R_AF:bool = unknown "AF is undefined after xor":bool
R_ZF:bool = true
R_PF:bool = true
R_OF:bool = false
R_CF:bool = false
R_SF:bool = false
```

Listing 2.7: `XOR EAX, EAX` lifted to BAP BIL.

2.4.2.2.2 iltrans For user-prescribed transformations, the *iltrans* utility can modify BIL code into several different forms. This utility can create Abstract Syntax Tree (AST), Control-Flow Graph (CFG) and many other outputs. Numerous transformations are possible as a series of layers to refine the BIL for a given analysis.

Chopping is a transformation which reduces a program down to only the BIL statements that affect a sink node (destination) for a given source node. Other transforms perform the removal of particular undesirable BIL such as dead (unreachable) or indirect (unsolvable) code.

2.4.2.2.3 topredicate The *topredicate* command can transform a program into a logical expression. This expression, known as the *weakest precondition*, defines the minimal circumstance where the program is certain to finish in a predicted state. This tool integrates with Satisfiability Modulo Theory (SMT) solvers in order to compute the expressions created by *topredicate*.

2.4.2.2.4 ileval The *ileval* utility enables *concrete evaluations* to execute BIL code natively instead of requiring recompilation into higher-level languages. Variables added to the BIL program can determine how a program would execute. Flags, registers and memory can also be set at any point in the BIL code to simulate specific conditions. For example, if evaluating a suspected anti-debugging technique in a program, *tainting* the memory address checked by the windows `isDebuggerPresent()` function could affect the execution of the program (see Listing 2.8). *ileval* can execute the tainted BIL and determine the result.

```
1 // initialize segment register base address
2 R_FS_BASE:u32 = 0x0:u32
3 mem:?u32 = mem:?u32 with [R_FS_BASE:u32 + 0x30:u32, e_little]:u32 =
        0xdeadbeef:u32
4 // taint fs:[30h] + 2 = 1
5 mem:?u32 = mem:?u32 with [0xdeadbeef:u32 + 2:u32, e_little]:u8 = 0x1
    :u8
```

Listing 2.8: BIL code to taint the `isDebuggerPresent()` byte.

2.4.2.3 Limitations

BAP has several limitations. The main limitations outlined in the documentation are (a) only x86 and x86-64 processors supported; (b) does not support analysis of

non-deterministic behaviors; (c) user-mode only; and (d) does not support floating-point instructions. Additionally, BAP does not support instruction sequences that form cycles. BAP will "unroll" a loop n times, but prior knowledge of n is required for correct analysis. Furthermore, while BAP can analyze Windows PE binaries, it does not function in Windows environments. BAP utilities can only be executed on Linux or Mac OSs.

2.5 Related Work

The following subsections describe research projects that have attempted to improve performance of RCE activities [35, 46, 75] and projects that have addressed similar problems in SE activities [48, 73].

2.5.1 Instruction Trace Pattern Matching

Instruction Trace Pattern Matching (ITPM) is an automated approach to detecting anti-debugging [75]. It is designed to search dynamic instruction traces for instances of anti-debugging patterns. First, to improve detection, a *trace refiner* scrubbed traces to remove unnecessary or obfuscating instructions. Once scrubbed, heuristic rules attempt to match the traces.

Xie, et al. conducted tests of the ITPM approach using 25 rules in four categories of anti-debugging: API calls, OS flags, magic strings and others [75]. Experiments processed 768 malware samples with a total detection rate of nearly 39%.

ITPM has two key characteristics that should be investigated: static vs. dynamic analysis and heuristic dependence. First, ITPM is a dynamic tool that relies on complete instruction traces. This in turn increases the probability of infection and requires the protection of a VM or emulator. As a result, ITPM must be conscious of execution safety. Static tools do not have this issue. Second, large instruction sets can subvert instruction-level analysis. Instruction matching rules must capture all possible combinations of instructions for a technique. If ITPM used IR, it could eliminate much of the confusion

caused by large instruction sets. This would allow for more concise rules that covered many different implementations.

2.5.2 Divergence Detector

Anti-VM techniques share many of the characteristics and motivations as anti-debugging techniques. Divergence Detector is a system for detecting such anti-VM techniques in malicious programs [46]. This method followed the principle that at any time during execution, a program can only execute one anti-VM check.

Divergence Detector is a system built upon three common malware analysis VMs: QEMU, Xen and Bochs. Each VM is loaded with the same guest OS and sample program. When ready, the system executes the malware sample, outputs an instruction trace and rolls back to a pre-test state. Divergence Detector compared traces and noted execution differences as divergences. Wherever the execution paths differ, VM checks are present. To eliminate uncertain false-positives, the process repeats several times to remove non-deterministic divergences from analysis.

When tested, Divergence Detector was capable of detecting instances of anti-VM techniques in malware samples. Hsu, et al. describe several trials where a divergence occurs in one of the VM environment but not the others [46]. Analysis of uncertainty reduction in the system revealed as the number of experiment rounds increased, the number of false detections decreased. In the test program, false divergences followed the probabilistic model very closely and disappeared after seven rounds.

2.5.3 Static and Dynamic Analysis

Eisenbarth, et al. describe a system for automating portions of RCE tasks by static and dynamic analysis [35]. Their approach first employs mathematical *concept analysis* to analyze binary relationships and derive a framework of all concepts used in a particular context called a concept lattice. This concept lattice then targets the dynamic analysis to identify sub-programs used for a set of features. Finally, static analysis separates essential

and non-essential sub-programs to focus human reverse engineering activities. In their experiments analyzing two web browsers, the researchers were able to reduce the search space of subprograms requiring further investigation by 98%.

This approach depends on the expertise of a human reverse engineer. If the reverse engineer lacks the necessary skills, the analysis will stall. Using an ES would enhance this process by introducing expert knowledge into the automated analysis.

2.5.4 Plug-ins for Popular Debuggers

There are several popular debuggers, in use today, that allow the addition of plug-ins to extend their functionality [45]. Debuggers such as OllyDbg, Immunity, IDA Pro and WinDbg all have anti-debugging plug-ins available[1, 2, 11]. These plug-ins are created by individuals and small teams in the debugger user community. They rely on heuristic analysis to detect API calls used by anti-debugging techniques. Some plug-ins are in active development, others have terminated. All plug-ins for these debugger offer little or no documentation.

2.5.5 The Varlet Analyst

Database (DB) reverse engineering has issues very similar to those of RCE. DB schemas themselves often do not reveal the purpose or use cases for the DB. Schemas, code and documentation must be compiled and analyzed to reveal a design specification for a re-engineered system [48]. The *Varlet Analysis* is a knowledge-based, semi-automatic approach to improve performance of DB reverse engineering activities.

The Varlet Analysis attempted to combine automatic RE analysis with customized domain knowledge to produce additional hypotheses about the system. Generic Fuzzy Reasoning Nets (GFRN) provided an abstract graphical framework for capturing specific domain knowledge. The GFRN and automated results were provided to an inference engine which generates additional hypotheses for addition into a working logical schema.

Confirmed hypotheses were used to infer a logical schema for the final translated conceptual design.

In addition to its focus on DB reverse engineering, the Varlet Analysis depends on application source code, schemas and domain experts to complete a conceptual design. Typical RCE activities do not benefit from these resources. However, an iterative automated analysis to human hypothesis verification loop could have use in analyzing code samples that resist other forms of analysis.

2.5.6 RODS and HASTI: Software Engineering Cognitive Support

Software Engineering (SE) is another area of computer science where tasks are difficult to automate and human cognition is critical. The purpose of the RODS framework is to reduce the complexity of code samples to improve the developer's cognition [73]. The HASTI framework works to augment the developer's cognition of SE tasks by modeling the elements and interactions affecting cognition. Together RODS and HASTI can improve developer performance by aiding cognition.

The RODS framework was designed as an application of the following principles: "task reduction", "algorithmic optimization", "distribution", and "specialization". Task reduction removes redundant tasks and replaces complicated tasks with simpler versions of the same task. Algorithmic optimization can improve efficiency and understandability by reducing the computational complexity of algorithms. Distribution acts to offload knowledge and mental states from the developer to a computer by artifact management and computational assistance. Finally, specialization aids the developer by offering task specific tools that more general tools lack. An example of specialization in computer science is using a language specific integrated development environment instead of a generic text editor for software development.

HASTI describes the models and methods used for analysis: "hardware models", "agent models", "specialization hierarchy, "task taxonomy", and "interaction abstraction

layer". The hardware model maintains specific facts related to the system hardware such as processing and memory limitations. The agent model relates individual "goal-focused tasks" to specific application components and code. The specialization hierarchy identifies relative task complexity by associating specific solution processes to development tasks based on how well the solution applies to the activity. A task taxonomy is used break down large complicated tasks into smaller tasks of known complexity. Finally, an interaction abstraction layer helps the developer by simplifying the interface between the software and hardware components.

SE and RCE each have many activities that affect human cognition. A framework resembling RODS and HASTI could provide cognitive assistance to the human preforming RCE activities. Particularly, RODS-like functionality could work to de-obfuscate disassembled code or associate unknown segments of code to known samples. A framework resembling HASTI could provide assistance by tracking and maintaining system and application details, freeing the human reverse engineer to focus on other details.

III. Methodology

3.1 Introduction

This chapter defines the methodology for implementation and testing the Rule Engine Detection by Intermediate Representation (REDIR) system for automating the static detection of obfuscated anti-debugging techniques in software samples. The goals and hypothesis behind this research are given in Section 3.2. Section 3.3 describes the design of the REDIR system. Section 3.4 provides the REDIR system architecture, its hardware and software specifications, and the development environment used. The test corpus employed for this research is specified in Section 3.5. In Section 3.6, the details of each test case are provided. Lastly, the results of pilot experimentation are offered in Section 3.7.

3.2 Goals and Hypothesis

The goal of this research is to demonstrate that (a) an Intermediate Representation (IR) based system can detect common analysis evasion techniques in program samples; (b) a rule-based Expert System (ES) can do the high-level matching required to reduce the search space; and (c) this method is resistant to common obfuscation techniques. To achieve this goal, the Data/Frame sensemaking theory guides the process of developing minimal starting information into complete anti-debugging detections.

The following hypothesis drives this research. Most anti-debugging techniques begin at some calculated or retrieved value α and end at a control-flow decision β. In program P, a rule R that searches for α and β can lead to the creation of a sub-program $C = \{\alpha...\beta\}$ for instance $T(R, \alpha, \beta)$ of anti-debugging technique R. If C is valid in P, then C instrumented with additional data can replicate non-debugging (C_{nd}) and debugging conditions (C_d). Evaluation of C_{nd} and C_d creates boolean values E_{nd} and E_d respectively. If comparison of

E_{nd} and E_d result in an inequality, then the data that replicated the debugging conditions caused the divergence. The divergence confirms the detection of $T(R, \alpha, \beta)$ in P.

Figure 3.1 depicts the REDIR concept through the Data/Frame sensemaking process. Frames are "constructed" with the detection of α and β. First, "questioning" creates the sub-program C that provides for "elaboration" to create instrumented sub-programs C_{nd} and C_d are created. Then, "questioning" resumes by evaluating C_{nd} and C_d to create E_{nd} and E_d. Finally, "questioning" E_{nd} and E_d to determine an inequality confirms the detection of the anti-debugging technique.

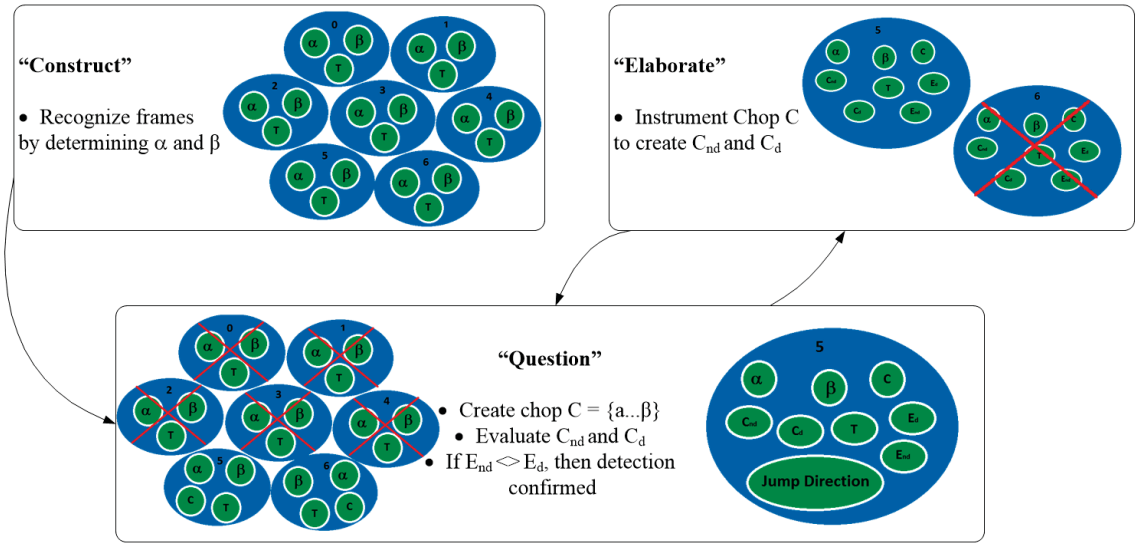

Figure 3.1: REDIR concept through the Data/Frame sensemaking process.

3.3 Design

3.3.1 Rule Engine Detection by Intermediate Representation

The REDIR system is a debugger plug-in written in the Java language. It was designed to process 32-bit Windows executables and identify instances of anti-debugging. The integration with the debugger disassembly view provides users visual identification of the anti-debugging instance.

3.3.2 Expert System Selection - Rule Engine

Numerous ES technologies are presented in this research. Each was evaluated for use in a Reverse Code Engineering (RCE) tool. The selection of a rule-based system was made based on its intended purpose, comparability with the other system components and usability. Readers should consider the selection of a rule-based system as a best-fit decision, not an endorsement or recommendation. This research did not perform in-depth testing of all possible ES technologies. Future work in the conclusion of this document will describe related research considered early in the process for implementation using the other ES technologies.

The rule engine selected for REDIR was JBoss Drools [5]. Drools was selected because it is a Java-based rule engine which supports object-oriented rule processing with an intuitive syntax.

Rules written for REDIR are high-level patterns of IR. These patterns provide a minimal representation of the technique to build the detection on. Based on the Data/Frame sensemaking model, each rule activation by the rule-engine (referred to as "frame" for the remainder of this document) serves as the starting point for more in-depth analysis. First, Listing 3.1 depicts the rule used to detect instances of the PEB!IsDebugger technique. This implementation is only concerned with accessing the `FS` segment register where the byte resides and any conditional jump. Then, Listing 3.2 shows the rule for detecting RDTSC-based timing techniques. This rule checks for two unique `rdtsc` calls, following the same register from each (either `EAX` or `EDX`). Finally, Listing 3.3 provides the rule for testing the MOV SS technique. Similar to the PEB!IsDebugger rule, this rule only looks for two statements; data stored to the Stack Segment (SS) register and a conditional jump.

```
rule "PEB_isDebuggerFlag_1"
  when
    $source : Statement(
      il matches  ".*R_FS_BASE:u32.*")
    $sink : Statement(
      il matches  ".*cjmp.*")
  then
  //Activation returns to main program
end
```

Listing 3.1: Drools rule for the PEB!IsDebugger anti-debugging technique.

```
rule "rdtsc_timing_1"
  no-loop true
  when
    //Step 1 - Obtain base address of FS register
    $source  : Statement(
      $il_1 : il,
      $addrAsInt_1 : addrAsInt,
      asm matches  ".*rdtsc.*")
    $2 : Statement(
      $addrAsInt_2 : addrAsInt,
      $il_2 : il,
      asm matches  ".*rdtsc.*")
    //Verify $1 and $2 use same registers.
    eval ($il_1 == $il_2)
    //Verify addresses are different. Prevents duplicate returns.
    eval ($addrAsInt_1 != $addrAsInt_2)
    //Verify 1st and 2nd timing checks are different statements
    eval ($source  != $2)
    $sink : Statement(
      il matches  ".*cjmp.*")
  then
  //Activation returns to main program
end
```

Listing 3.2: Drools rule for the RDTSC Timing anti-debugging technique.

```
rule "Mov_SS_1"
  no-loop true
  when
    $source  : Statement(
      il matches  ".*R_SS:u16 =.*")
    $sink : Statement(
      il matches  ".*cjmp.*")
  then
  //Activation returns to main program
end
```

Listing 3.3: Drools rule for the MOV SS anti-debugging technique.

3.3.3 Intermediate Representation

The hidden nature of anti-debugging techniques drove the decision to use an IR technology. IR tools can illuminate the hidden aspects of anti-debugging with extremely beneficial capabilities. However, like all tools, IR has its limitations. To accept the added functionality, this research also had to accept the limitations of the selected IR implementation.

The IR technology selected for this project was Binary Analysis Platform (BAP) 0.7 for the following reasons [24]. First, BAP is in active development and has an active user group for support. Next, BAP has an abstracted Intermediate Language (IL) that offers easily analyzable instruction semantics. Finally, BAP offers concrete execution of IL. This capability offered to not only detect instances of anti-debugging, but also determine the jump direction caused by the detection of the debugger.

To facilitate the use of BAP for evaluating programs, each frame from the rule engine contains the *source* and *sink* nodes that mark the beginning and end for a sub-program in BIL known as a *chop*. An instrumentation process adds additional variables and tainted values to the chopped program for evaluation. Each anti-debugging technique requires a different unique instrumentation. Listing 3.4 demonstrates an example of a minimum instrumentation and Algorithm 1 depicts a generic instrumentation process.

```
goal:bool = true //InitGoalBooleanString
// InitMemString ...
// Chop C ...
// BB_ERROR replaced with appropriate targets
cjmp ~R_ZF:bool, "JMP", "NOJMP"

label NOJMP     //NOJMPLabelString
goal := false   //UpdateGoalBooleanString
label JMP       //JMPLabelString
halt goal
```

Listing 3.4: Example of a minimum instrumentation (chopped code omitted).

Algorithm 1 Generic Instrumentation Algorithm

```
1:  procedure INSTRUMENT(Chop C, Boolean tainted)
2:      result.append(InitGoalBooleanString)
3:      result.append(InitMemString)
4:      result.append(C)
5:                                          ▷ Replace error jump targets with appropriate targets
6:      result.replaceFirst("BB_ERROR","JMP")
7:      result.replaceLast("BB_ERROR","NOJMP")
8:      result.append(NOJMPLabelString)
9:      result.append(UpdateGoalBooleanString)
10:     result.append(JMPLabelString)
11:     if tainted == true then
12:         result.replace(InitMemString, TaintedMemString)
13:     end if
14:     return result
15: end procedure
```

3.3.4 Algorithm

Following initialization by the rule-engine, each frame $T(R, \alpha, \beta)$ develops in the Data/Frame sensemaking model by *questioning*, *elaborating* and *evaluating* the frame. The REDIR algorithm *questions* by attempting to add chop $C = \{\alpha...\beta\}$ to the frame with the BAP *iltrans* utility (see Algorithm 2). Next, if chopped successfully, instrumentation *elaborates* C to form C_{nd} and C_d versions of the sub-program for evaluation. The final step uses the BAP *ileval* command (beginning on line 13) to *evaluate* C_{nd} and C_d. If the evaluation results, E_{nd} and E_d respectively, are not the same, this shows the simulated debugging condition data affected the outcome of the program and confirms $T(R, \alpha, \beta)$ as an anti-debugging instance.

3.3.5 Debugger Interface

The DigR debugger hosts the REDIR plug-in and provides access to the executable for analysis as well as the architecture for information display [4]. When active, REDIR presents a table of frames. For each frame, REDIR lists important information such as the source and sink nodes, chop validity, detection, and jump direction. Additionally, when

Algorithm 2 REDIR Algorithm

```
1:  procedure GO(Program P)
2:      bil ← BAP.toil(P)
3:      ast ← BAP.iltrans(bil)
4:      ruleEngine.load(ast)
5:      F ← ruleEngine.fireAllRules()
6:      for f ∈ F do
7:          Chop C ← BAP.iltrans(ast, f.α, f.β)
8:          if valid(C) then
9:              C_d ← instrument(C, True, f.technique)
10:             C_nd ← instrument(C, False, f.technique)
11:             E_d ← BAP.ileval(C_d)
12:             E_nd ← BAP.ileval(C_nd)
13:             if E_d <> E_nd then
14:                 f.detected ← true
15:             else
16:                 f.detected ← false
17:             end if
18:         end if
19:     end for
20:     return F
21: end procedure
```

selected in REDIR, the disassembly view will update to highlight the instructions used by the technique.

3.3.6 Design Considerations

Based on previously described limitations of the BAP framework (Section 2.4.2.3), REDIR has two significant restrictions. Future releases of the BAP framework may mitigate these limitations.

3.3.6.1 Cycles

Anti-debugging techniques that form cycles such as loops are not analyzable in the BAP Framework. Subsequently, REDIR cannot detect these techniques.

3.3.6.2 Operating System Compatibility

BAP is not compatible with the Windows Operating System (OS). BAP operates only in Linux and Mac environments. DigR is a Windows debugger. To facilitate using BAP with DigR a bridge was required. As REDIR was already Java-based, a simple Java-

based proxy interacted with REDIR via Java Remote Method Invocation (RMI). This proxy received input from REDIR, executed the desired BAP program and returned the result to REDIR.

3.4 Implementation

REDIR and the BAP proxy executed within connected Virtual Machines (VMs). As depicted in Figure 3.2, a single Windows 8 computer with VMWare Workstation 9.0 hosted each of the VMs. For this system, DigR executed inside a Windows 7 VM and the BAP Framework and proxy inside an Ubuntu Server 12.04 VM. REDIR itself is a plug-in for the DigR debugger. As a plug-in, REDIR had access to show highlight anti-debug techniques in the DigR Disassembly View.

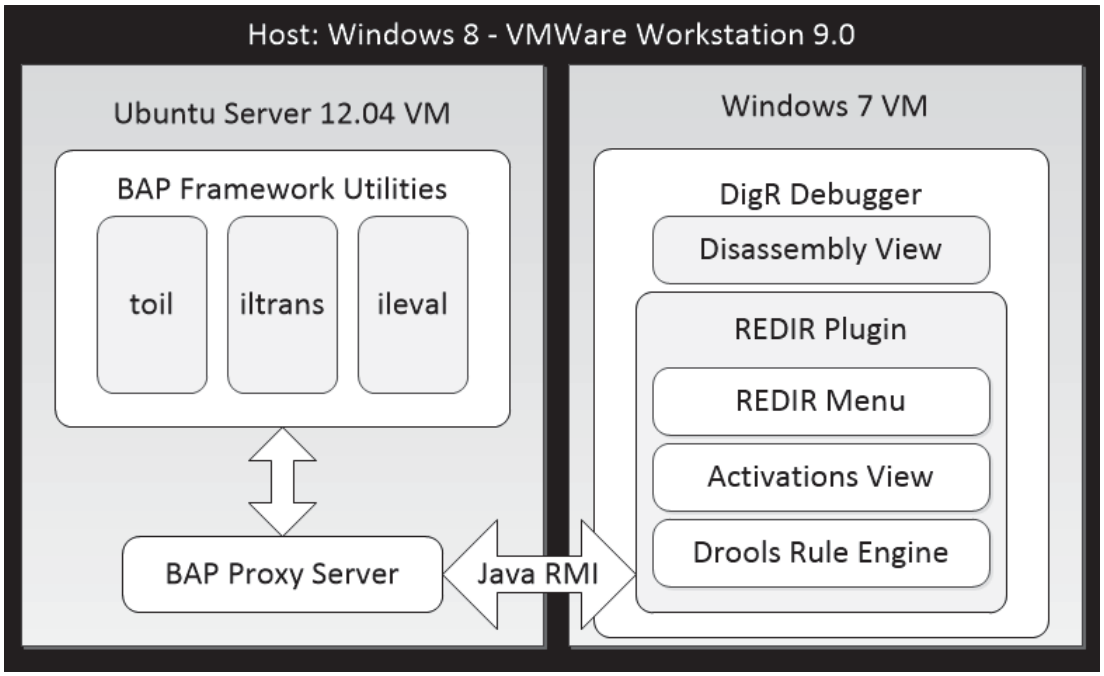

Figure 3.2: REDIR system configuration.

3.4.1 Hardware and Software Specifications

- Host: Lenovo Y500 laptop computer

  - OS: Windows 8
  - CPU: Intel Core i7-3630QM
  - Random Access Memory (RAM): 16 Gigabyte (GB)
  - Key Software: VMWare Workstation 9.0
- DigR Host and Development VM
  - OS: Windows 7
  - CPU setting: 2 processors, 2 cores per processor
  - RAM setting: 4 GB
  - Hard disk setting: 60 GB
  - Key Software: DigR Debugger, Eclipse - Kepler Release
- BAP Host VM
  - OS: Ubuntu Server 12.04
  - CPU setting: 1 processor, 1 core per processor
  - RAM setting: 1 GB
  - Hard disk setting: 20 GB
  - Key Software: BAP Framework

3.4.2 Development Environment

REDIR and the BAP Proxy were developed using Eclipse: Kepler Release. Plug-in development occurred within the Windows 7 VM described in the previous section. Proxy development took place on an Ubuntu 12.04 VM with an identical BAP configuration.

3.5 Testing Methodology

Unfortunately, it is very difficult to find anti-debugging technique samples that will disassemble correctly, are unencrypted, and guaranteed to exhibit the desired behavior. For these reasons, synthetic program samples modeled real-world malware anti-debugging

implementations. All samples used the Intel Architecture, 32-bit (IA-32) Microsoft Macro Assembler (MASM) assembly syntax and compiled with the MASM assembler. Each simple program attempts debugger detection and prints either "Debugger found" or "No debugger found" based on the detection result. For each anti-debug technique, one of each of the obfuscations disguised the technique. Due to the similarities among anti-debugging techniques, not all known anti-debugging techniques were required for testing. Many techniques employ the same overall strategy; as a result, their associated clues and instrumentations would only be slightly different and no less solvable. For testing REDIR, the techniques are broken down into representative categories as follows:

1. OS Flags - Represented by the Windows PEB!IsDebugger byte. See Section 2.2.3.3.1.
2. Timing - Represented by the RDTSC Timing technique. See Section 2.2.3.3.2.
3. Interrupt Handling - Represented by the MOV SS technique. See Section 2.2.3.3.3.

For each anti-debug technique, each of the following obfuscations will be applied to form a matrix of anti-debug technique/obfuscation samples.

1. No Obfuscation
2. Dead Code Insertion
3. Register Reassignment
4. Code Transposition
5. Instruction Substitution
6. Conditional Code Obfuscation

Obfuscation numbers two through five are forms of layout obfuscation (see Section 2.2.3.4.1). Obfuscation six is a form of conditional code obfuscation (see Section 2.2.3.4.2).

3.6 Experiment Design

To adequately test the REDIR system, each of the program samples must be analyzed in the debugger. For each sample, the debugger will load the executable and launch the REDIR process. There are four measures to describe the performance of the system.

1. Source/Sink Identification - Did REDIR find the source and sink for the implemented anti-debug technique? This evaluation infers the correct operation of the frame construction mechanism. Without the correct source and sink nodes, REDIR will not find the implemented anti-debug technique.
2. Chop Identification - Did REDIR highlight the correct instructions in the disassembly? This analysis is important because an incorrect chop could lead to an incorrect positive or negative detection result.
3. Anti-debug Technique Identification - Did REDIR find the implemented anti-debug technique? This evaluation is most important as it demonstrates the feasibility of the system.
4. Jump Direction - Did REDIR predict the correct jump direction based on a simulated anti-debug condition? During preliminary development incorrect or non-identification of the jump direction strongly implied a critical error that could invalidate one or more of the previous evaluations. Correct identification during this evaluation reinforces the previous evaluation results.

Due to the limitations imposed by the system design, execution time for the REDIR plug-in is not an informative metric.

The following list describes six test cases that were used detail the evaluation of REDIR. These cases were selected because together they fully test the REDIR system under all the available anti-debug techniques and obfuscations. Additionally, for each technique, they demonstrate how a single strategy can detect multiple implementations.

The remaining twelve test cases were tested following the same methodology. Their results are summarized in Chapter 4.

3.6.1 Test Case #1: PEB!IsDebugger/No Obfuscation

The purpose of this first test case is to demonstrate REDIR with a non-obfuscated anti-debug technique. The code sample for this test case is the same as the sample shown in Listing 2.2. Evaluation of REDIR's analysis should reveal detection of the PEB!IsDebugger technique comprised of four lines of code beginning with the `FS` register access and ending with a conditional jump. Additionally, based on the use of the `test eax,eax` and `jne DebuggerFound` instructions, analysis should conclude that using a debugger will cause taking the conditional jump instead of falling through to the next instruction.

3.6.2 Test Case #2: RDTSC Timing/Dead Code Insertion

The second test case demonstrated that REDIR could ignore meaningless code inserted between important instructions. As shown in Listing 3.5, this program has unnecessary `nop` instructions added. REDIR should reveal detect the RDTSC Timing technique in eight lines of code beginning with the first `rdtsc` instruction and ending with the conditional jump. The `nop` instructions should be ignored. The final two instructions, `cmp eax, ecx` and `ja DebuggerFound`, would cause this program to jump is sufficiently delayed. REDIR should detect a debugger will cause the program to choose taking the conditional jump.

```
1  rdtsc
2  nop
3  xor ecx, ecx
4  nop
5  add ecx, eax
6  nop
7  rdtsc
8  nop
9  sub eax, ecx
10 mov ecx, 0FFFh
11 nop
12 cmp eax, ecx
13 ja DebuggerFound
```

Listing 3.5: IA-32 implementation example of RDTSC Timing detection technique with dead code obfuscation applied.

3.6.3 Test Case #3: MOV SS/Register Reassignment

Test case three demonstrates how BAP enables REDIR to follow important data despite moving it between registers. Listing 3.6 shows how this program attempts to conceal testing the trap flag. Here, the program grabs the bit from the stack and places it in the `bx` register. Next the value is moved to the `cx` register before eventually being tested on line six. REDIR should detect exactly six lines of code for this technique starting with the `pop ss` instruction and ending with the conditional jump. The initial `push ss` instruction is not part of the technique, it only preserves the original value of the `SS` register. If the trap flag is set, the last two instructions, `test cx, 1` and `jne DebuggerFound`, would cause this program to jump. REDIR should detect a debugger will cause the program to choose taking the conditional jump.

```
1 push ss
2 pop ss
3 pushfd
4 mov bx, word ptr [esp+1]
5 mov cx, bx ; reassign register
6 test cx, 1
7 jne DebuggerFound
```

Listing 3.6: IA-32 implementation example of the MOV SS detection technique with register reassignment obfuscation applied.

3.6.4 Test Case #4: PEB!IsDebugger/Code Transposition

As depicted in Listing 2.5, the purpose of this test case is to demonstrate REDIR's performance with unordered code. REDIR should detect the PEB!IsDebugger technique in seven lines in the order (12, 13, 6, 7, 3, 4, 9) beginning with the `FS` register access and ending with a conditional jump. Just like test #1, this test case employs the `test eax,eax` and `jne DebuggerFound` instructions. As a result, REDIR should show that debugging would cause following the conditional jump.

3.6.5 Test Case #5: RDTSC Timing/Instruction Substitution

The purpose of test case #5 is to show that REDIR can detect a technique when alternative instructions are used. Based on Listing 2.3, this example shows the replacement of several instructions while maintaining the original functionality (see Listing 3.7). REDIR should detect the RDTSC Timing in exactly nine lines of code starting with the first `rdtsc` instruction and ending with the conditional jump. The `push/pop` combination replaces the original `add` instruction and a `sub` instruction replaces the `cmp` instruction to invert the conditional jump evaluation. The conditional jump has also been replaced. The `jnl` instruction replaces the `ja` instruction. The first jumps if the tested value is above the threshold. The second jumps if the tested value is not less than the threshold. Finally, the jump target has been replaced. Instead of jumping when being debugged, this sample will jump if not debugged. REDIR should detect the conditional jump will not be taken if being debugged.

```
1 rdtsc
2 xor ecx, ecx
3 push eax
4 rdtsc
5 pop ecx
6 sub eax, ecx
7 mov ecx, 0FFFh
8 sub ecx, eax
9 jnl NoDebuggerFound
```

Listing 3.7: IA-32 implementation example of RDTSC Timing detection technique with instruction substitution obfuscation applied.

3.6.6 Test Case #6: MOV SS/Opaque Predicate

The purpose of the final test case is to show that REDIR can employ concrete evaluation to decipher a complex obfuscation and find an anti-debug technique. The implementation of the MOV SS technique shown in Listing 3.8 begins on line 3 at the `pop ss` instruction and terminates at line 32 with the `jne` instruction. Since all the selected instructions between carry data from beginning to end, 30 lines of code represent the entire MOV SS technique. Similar to the first four test cases, this test case should take the conditional jump if detecting a debugger.

```
xor ebx, ebx
push ss
pop ss
pushfd              ; push EFLAGS
xor eax, eax
add ax, x
add ax, y
imul ax, ax
push ax             ; push (x+y)^2
xor eax, eax
mov ax, x
imul ax, ax
push ax             ; push x^2
xor eax, eax
mov ax, y
imul ax, ax
push ax             ; push y^2
xor eax, eax
xor ebx, ebx
mov ax, x
mov bx, y
imul ax, bx
imul ax, 2          ; ax = 2xy
pop bx              ; bx = y^2
add ax, bx          ; ax = 2xy + y^2
pop bx              ; bx = x^2
add ax, bx          ; ax = x^2 + 2xy + y^2
pop bx              ; bx = (x+y)^2
cmp ax, bx          ; always evaluates true: ax == bx
jne fake            ; never jumps
test word ptr [esp+1], 1 ; test trap flag
jne DebuggerFound
```

Listing 3.8: IA-32 implementation example of the MOV SS detection technique with opaque predicate obfuscation applied.

3.7 Pilot Experiment

Initial experimentation with the BAP framework and the Drools rule engine led to the creation of the REDIR system. However, the first iteration design of the REDIR system had several flaws which contributed to the design decisions made for the final version. The issues with found during pilot experimentation are due to rule engine processing of anti-debug techniques, BAP deployment and TF support.

3.7.1 Rule Engine Processing

The pilot design depended heavily on the rule engine for anti-debug detections. This design utilized complete anti-debug technique heuristic patterns in the IR. The drawback of this approach was that rules were fragile and not provable. It was easy to write test programs which could evade detection. Additionally, this design required the rule engine have the ability to follow the flow of execution. Additional processing steps helped establish dependence and track the flow of execution since instructions without common data were not likely anti-debug implementations.

To improve the design, two improvements were implemented. First, the rule engine's role was modified. Instead of looking for complete techniques, the rules were relaxed to find sections where a technique could be present. The new rules only look for the few IR statements that must occur in the anti-debugging technique sought. Other common, but not required, IR statements are ignored by the rules. Chops could be created based on the rule engine detections that contained pertinent IR. The second improvement was the inclusion of concrete evaluation. This improvement allows rule engine detections evaluations in the IR to prove the technique detection.

Because of the design changes, several improvements were realized. The improved REDIR design was much faster. Execution times improved from over one minute down to less than two seconds. This improvement was largely due to reduced rule engine processing. Additionally, the new design was more robust. Technique implementations that bypassed detection before became detectable. Instrumented chops derived from the simple rule detections were easier to evaluate than entire programs with the rule engine's complex heuristic rules.

3.7.2 Binary Analysis Platform Deployment

As stated in Section 3.3.6.2, the BAP framework is only compatible with Linux and Mac environments. Meanwhile, the DigR debugger is a Windows only tool. Several

deployment attempts tried to place both tools on the same platform. However, no method was found to install both tools on the same computer. Two attempts were made to use a Linux emulator for BAP deployment. First, BAP deployment was attempted in *Cygwin* [3]. Unfortunately, unidentified secondary dependencies were not available in the Cygwin environment; BAP installation could not be completed. Installation attempts with Minimalist GNU for Windows (MinGW) faced the same dilemma [8]. All attempts to install BAP in Windows were unsuccessful. Consultation with the BAP User Group revealed no concerted effort to deploy BAP in a Windows environment.

At the time of this experimentation, an Ubuntu 12.04 VM hosted the BAP framework and a Windows 7 VM hosted DigR. The next available strategy, the client-server model described in Section 3.4, was quickly implemented and tested. This design provided the desired performance and usability without the need for additional configuration changes. This design proved to be satisfactory and remains part of the REDIR system.

3.7.3 Trap Flag Support

BAP 0.7 does not support the TF by default. To support the MOV SS technique the capability to track the TF was required. The BAP source code was modified for this research to allow analysis involving the TF. With the assistance of the BAP Users Group, I was able to add the feature and implement the MOV SS anti-debug technique detection.

3.8 Methodology Summary

The purpose of this chapter was to introduce the REDIR system and the methodology behind its implementation. First, its design, development and implementation details were given. Next, the test methodology and its rationale was offered. Then, the experiment design with specific test case descriptions was provided. Finally, pilot experimentation results contributing to the design decisions described in this chapter. The next chapter describes how testing was conducted and the results of that testing.

IV. Experiment Results

4.1 Introduction

This chapter details the test results and test analysis for the Rule Engine Detection by Intermediate Representation (REDIR) system. First, overall test evaluation and analysis are offered in Section 4.2. Next, in Section 4.3, detailed test reports describe each of the six test case results. Finally, Section 4.4 provides analysis of the design and implementation of the REDIR system.

4.2 Evaluation and Analysis

The purpose of testing the REDIR plug-in was to determine if the tool was capable of detecting anti-debugging techniques in obfuscated code. Testing followed by initializing the REDIR plug-in for each test program and manually analyzing each result (see Section 3.5). Screen captures recorded the analysis results displayed in the DigR window for each attempt. Success was evaluated by (a) correct identification of instruction lines used by the technique; and (b) correct determination of jump direction.

The REDIR plug-in analyzed each of the 18 anti-debugging/obfuscation sample programs. In each case, the tool created multiple frames during the analysis. Many of the generated frames were invalid and correctly discarded. Most programs correctly yielded only one valid frame. In all test cases, the REDIR correctly identified the technique and highlighted the set of instructions that affected the outcome of the program. REDIR was 100% effective for those 18 test cases. REDIR did not highlight irrelevant instructions with no bearing on the outcome. Table 4.1 summarizes the entire test corpus and detection results.

Table 4.1: Test results summary

Anti-debugging \ Obfuscation	None	Dead Code Insertion	Register Reassignment
PEB!IsDebugger	Detected Section 4.3.1	Detected	Detected
RDTSC Timing	Detected	Detected Section 4.3.2	Detected
MOV SS	Detected	Detected	Detected Section 4.3.3
Anti-debugging \ Obfuscation	**Code Transposition**	**Instruction Substitution**	**Opaque Predicate**
PEB!IsDebugger	Detected Section 4.3.4	Detected	Detected
RDTSC Timing	Detected	Detected Section 4.3.5	Detected
MOV SS	Detected	Detected	Detected Section 4.3.6

4.3 Detailed Test Case Analysis

The following subsections detail the results of six test cases. These test cases are representative of the larger test corpus and eliminate redundant explanations. The test results obtained from the remaining twelve test cases closely follow those presented below.

4.3.1 Test Case #1: PEB!IsDebugger/No Obfuscation

4.3.1.1 Test Summary

This test of the PEB!IsDebugger technique paired with no obfuscation demonstrated REDIR operation in the simplest case. The frame at index five in Figure 4.1 correctly identified the implemented technique. Figure 4.2 shows just the four original lines of code selected in the disassembly view. Aside from minor disassembly differences, this test case matches the original sample perfectly (see Figure 2.2).

4.3.1.2 Source/Sink Identification

The rule for this technique searches for a pair of instructions. The first a `FS` access and the second a conditional jump. The lines `0x401000` and `0x40100c` created by rule engine for the frame at index five correctly match to the `FS` access and a conditional jump. The remaining frames matched other instruction pairs and persisted temporarily to make a chop attempt.

4.3.1.3 Chop Identification

A chop for each of the frames captured by the rule engine was attempted. Beginning with the source and sink nodes `0x401000` and `0x40100c`, *iltrans* correctly created a chop for the frame at index five. As predicted, only the instructions that carry data from the source to the sink display in the disassembly view.

4.3.1.4 Anti-debugging Technique Identification

In this sample, exactly one implementation of the PEB!IsDebugger technique was implemented. The frame at index five correctly asserts the presence of the technique.

4.3.1.5 Jump Direction

Based on the result of the instrumented evaluation simulating the debugging condition, REDIR correctly identified that the presence of a debugger would cause this program to jump (to line `0x40101a`) instead of fall through to the next instruction (at line `0x40100e`).

Index	Rule	Source	Sink	Valid Chop?	Detected?	Tainted Jumps?
0	PEB_isDebuggerFlag_1	401000	4010d0	false		?
1	PEB_isDebuggerFlag_1	401000	4010c8	false		?
2	PEB_isDebuggerFlag_1	401000	4010b5	false		?
3	PEB_isDebuggerFlag_1	401000	4010a2	false		?
4	PEB_isDebuggerFlag_1	401000	40108f	false		?
5	PEB_isDebuggerFlag_1	401000	40100c	true	yes	true

Figure 4.1: Created frames during analysis of PEB!IsDebugger technique without obfuscation.

```
00400FFF
00401000    mov eax, large fs:[30h]
00401006    movzx eax, byte ptr [eax+2]
0040100A    test eax, eax
0040100C    jnz short loc_40101A
0040100E    push 40300Fh
```

Figure 4.2: Highlighted DigR disassembly view of PEB!IsDebugger technique without obfuscation.

4.3.2 Test Case #2: RDTSC Timing/Dead Code Insertion

This test of the rdtcs timing technique paired with dead code insertion demonstrated REDIR operation in the simplest case of obfuscation. The frame at index five in Figure 4.3 correctly identified the implemented technique. Figure 4.4 shows the eight original lines of code selected in the disassembly view. Between these lines, several nop instructions are skipped because they have no effect in this program. As in the previous case, this test case perfectly matches the original sample (see Figure 2.3).

Index	Rule	Source	Sink	Valid Chop?	Detected?	Tainted Jumps?
0	rdtsc_timing_1	401000	4010d0	false		?
1	rdtsc_timing_1	401000	4010c8	false		?
2	rdtsc_timing_1	401000	4010b5	false		?
3	rdtsc_timing_1	401000	4010a2	false		?
4	rdtsc_timing_1	401000	40108f	false		?
5	rdtsc_timing_1	401000	401016	true	yes	true

Figure 4.3: Created frames during analysis of the RDTSC Timing technique obfuscated by dead code insertion.

4.3.2.1 Source/Sink Identification

The rule for this technique searches for a pair of `rdtsc` instructions and a conditional jump. The lines `0x401000` and `0x401016` created by the frame at index five are correctly matched by the rule engine for the frame at index five correctly match to the first `rdtsc`

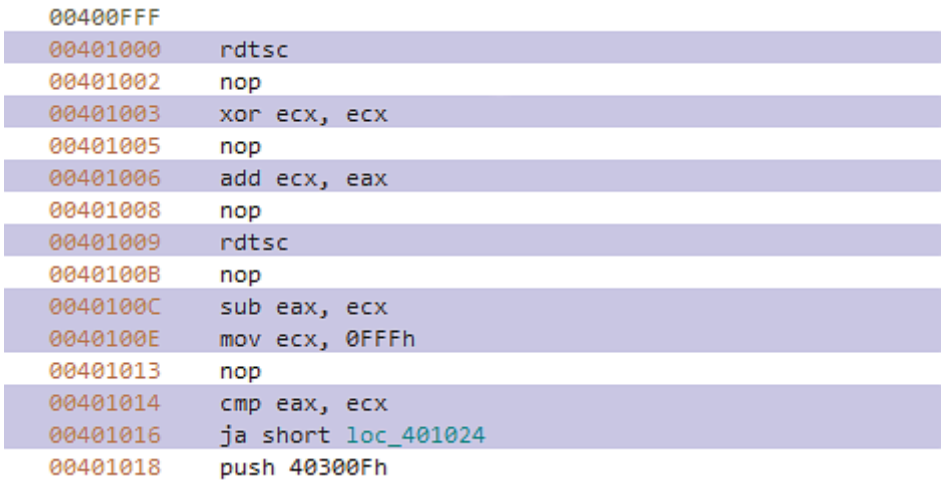

Figure 4.4: Highlighted DigR disassembly view of the RDTSC Timing technique obfuscated by dead code insertion.

instruction and a conditional jump. The remaining frames matched other instruction sets and continue on to the chop attempt.

4.3.2.2 Chop Identification

Again, the REDIR attempted a chop for each of the frames captured by the rule engine. Beginning with the source and sink nodes `0x401000` and `0x401016`, the frame at index five provided *iltrans* the correct source and sink nodes for a successful chop. By eliminating the `nop` instructions, REDIR only selected the participating instructions from the source to the sink. Frames zero through four were discarded because they did not lead to a valid chop.

4.3.2.3 Anti-debugging Technique Identification

This code sample contained exactly one implementation of the RDTSC Timing technique. Index five represents the frame that correctly identifies the instance of the technique.

4.3.2.4 Jump Direction

With a simulated delay in processing, the instrumented evaluation correctly identified that the presence of a debugger would cause this program to jump (to line `0x401024`) instead of fall through to the next instruction (at line `0x401018`).

4.3.3 Test Case #3: MOV SS/Register Reassignment

4.3.3.1 Test Summary

This test of the MOV SS technique paired with register reassignment demonstrated REDIR operation in a slightly more obfuscated condition. Again, the frame at index five in Figure 4.5 correctly identified the technique instance. Figure 4.6 shows the five lines of code selected in the disassembly view. However, unlike the previous cases, the DigR disassembly does not show the `SS` register on lines `0x401005` and `0x401006` as coded in Figure 2.4). Where a trained human reverse engineer may have missed the technique due to missing `SS` label, REDIR found the technique because it focused on the Intermediate Representation (IR) rather than the disassembled instructions.

Index	Rule	Source	Sink	Valid Chop?	Detected?	Tainted Jumps?
0	Mov_SS_1	401005	4010d0	false		?
1	Mov_SS_1	401005	4010c8	false		?
2	Mov_SS_1	401005	4010b5	false		?
3	Mov_SS_1	401005	4010a2	false		?
4	Mov_SS_1	401005	40108f	false		?
5	Mov_SS_1	401005	401014	true	yes	true

Figure 4.5: Created frames during analysis of the MOV SS technique obfuscated by register reassignment.

4.3.3.2 Source/Sink Identification

The MOV SS rule searches for source node that writes to the `SS` register and terminates at a conditional jump sink node. The frame at index five begins at line `0x401005` and ends

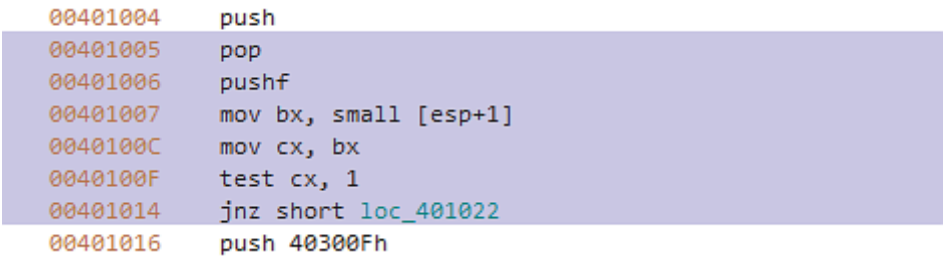

Figure 4.6: Highlighted DigR disassembly view of the MOV SS technique obfuscated by register reassignment.

at 0x401014; this is correct for the MOV SS rule. The remaining frames matched other instruction sets and continue on to the chop attempt.

4.3.3.3 Chop Identification

For each index identified by the rule engine, REDIR attempted to chop the program for that frame. Beginning with the source and sink nodes 0x401005 and 0x401014, index five provided the correct source and sink nodes for *iltrans* to create a successful chop. By eliminating the nop instructions, REDIR only selected the participating instructions from the source to the sink. The other frames started by the rule engine were discarded since they did not create valid chops.

4.3.3.4 Anti-debugging Technique Identification

This sample contains only one implementation of the MOV SS technique. Index five correctly identifies the instance of the technique.

4.3.3.5 Jump Direction

Simulating a debugging condition by setting the Trap Flag (TF), the instrumented evaluation identified that this program would take the jump (to line 0x401022) instead of fall through to the next instruction (at line 0x401016).

4.3.4 Test Case #4: PEB!IsDebugger/Code Transposition

4.3.4.1 Test Summary

The second test evaluated with the PEB!IsDebugger technique was paired with code transposition and begins to demonstrate that REDIR can find different instances of the same technique with only one rule. The frame at index five in Figure 4.7 shows the correctly identified technique. Figure 4.8 shows just the four original lines of code selected in the disassembly view. Aside from minor disassembly differences, this test case matches the original sample perfectly (see Figure 2.2).

Index	Rule	Source	Sink	Valid Chop?	Detected?	Tainted Jumps?
0	PEB_isDebuggerFlag_1	40101a	4010d0	false		?
1	PEB_isDebuggerFlag_1	40101a	4010c8	false		?
2	PEB_isDebuggerFlag_1	40101a	4010b5	false		?
3	PEB_isDebuggerFlag_1	40101a	4010a2	false		?
4	PEB_isDebuggerFlag_1	40101a	40108f	false		?
5	PEB_isDebuggerFlag_1	40101a	40100c	true	yes	true

Figure 4.7: Created frames during analysis of PEB!IsDebugger technique obfuscated by code transposition.

4.3.4.2 Source/Sink Identification

The rule used for source/sink identification in this case is the same as was used in the first evaluation (Section 4.3.1.2). The lines `0x40101a` and `0x40100c` created by the frame at index five correctly match the rule engine to a `FS` access and a conditional jump. The other frames matched with other instruction pairs and were evaluated in a chop attempt.

4.3.4.3 Chop Identification

A chop for each of the frames captured by the rule engine was attempted. Despite the source node's appearance after the sink node, *iltrans* correctly chopped the program from `0x40101a` to `0x40100c` for the frame at index five. In the disassembly view, the uninvolved

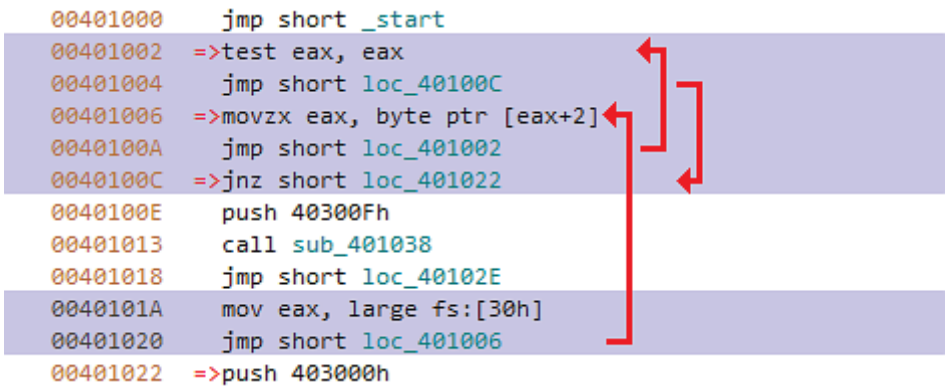

Figure 4.8: Highlighted DigR disassembly view of PEB!IsDebugger technique obfuscated by code transposition. Jumps are illustrated for clarity beginning after line `0x40101a` to the terminating instruction at `0x40100c`.

instructions are not selected since they do not affect the outcome of the anti-debugging technique. Without valid chops, the other frames started by the rule engine were discarded correctly.

4.3.4.4 Anti-debugging Technique Identification

As before, only one PEB!IsDebugger implementation was present in the program. The frame at index five correctly identifies that implementation.

4.3.4.5 Jump Direction

By simulating the debugging condition, REDIR correctly identified that using a debugger would cause this program to take the jump (to line `0x401022`) instead of falling through to the next instruction (at line `0x40100e`).

4.3.5 Test Case #5: RDTSC Timing/Instruction Substitution

4.3.5.1 Test Summary

Unlike the program in Figure 2.3, this sample uses different instructions to achieve the same functionality. The first timing value was pushed to the stack instead inserting it into a register. The `cmp` instruction was swapped out in favor of a `sub` instruction, and the

conditional jump was negated to force a debugging condition to not jump. Despite those changes, this test of the RDTSC Timing technique paired with instruction substitution demonstrated REDIR's resiliency to arbitrary obfuscation decisions. The frame at index five in Figure 4.9 identifies the correct anti-debugging instance. Figure 4.10 shows the nine original lines of code selected in the disassembly view. Again, line for line, this test case matches the original sample.

Index	Rule	Source	Sink	Valid Chop?	Detected?	Tainted Jumps?
0	rdtsc_timing_1	401000	4010d0	false		?
1	rdtsc_timing_1	401000	4010c8	false		?
2	rdtsc_timing_1	401000	4010b5	false		?
3	rdtsc_timing_1	401000	4010a2	false		?
4	rdtsc_timing_1	401000	40108f	false		?
5	rdtsc_timing_1	401000	401011	true	yes	false

Figure 4.9: Created frames during analysis of RDTSC Timing technique obfuscated by instruction substitution.

```
00400FFF
00401000    rdtsc
00401002    xor ecx, ecx
00401004    push eax
00401005    rdtsc
00401007    pop ecx
00401008    sub eax, ecx
0040100A    mov ecx, 0FFFh
0040100F    sub ecx, eax
00401011    jge short loc_40101F
00401013    push string_403000              "Debugger Found"
00401018    call near ptr 401030h
0040101D    jmp short _start
0040101F  =>push string_40300F              "No Debugger F...
```

Figure 4.10: Highlighted DigR disassembly view of RDTSC Timing technique obfuscated by instruction substitution.

4.3.5.2 Source/Sink Identification

As before, the rule engine searched for a pair of `rdtsc` instructions and a conditional jump. At index five, the lines `0x401000` and `0x401011` correctly match the first `rdtsc` instruction and a conditional jump. Indexes zero through four matched similar instructions and progressed on to the chop attempt.

4.3.5.3 Chop Identification

Beginning with the source and sink nodes `0x401000` and `0x401011`, the frame at index five provided the correct source and sink nodes to create a successful chop. Despite the obfuscation, *iltrans* was able to create a chop because the source and sinks provided by REDIR correctly bounded this anti-debugging instance. The other frames did not provide a valid chop and were discarded.

4.3.5.4 Anti-debugging Technique Identification

REDIR correctly identified the RDTSC Timing implementation in the code sample. The frame at index five correctly confirmed the instance of the technique.

4.3.5.5 Jump Direction

Unlike the previous test cases, this example intentionally chooses to fall through to the next instruction when sensing a debugger. With the simulated delay, the evaluation correctly identified that the program would fall through (to line `0x401013`) rather than jump (to line `0x40101f`).

4.3.6 Test Case #6: MOV SS/Opaque Predicate

4.3.6.1 Test Summary

This final test case paired the well-disguised MOV SS anti-debugging technique with a challenging opaque predicate obfuscation.

As before, the frame at index five in Figure 4.11 correctly identified the sought anti-debugging instance. Figure 4.12 shows the 30 lines of code that span the technique from

beginning to end. This view illustrates how other seemingly innocuous instructions can exist within a technique to create a more sophisticated anti-debugging instance.

Index	Rule	Source	Sink	Valid Chop?	Detected?	Tainted Jumps?
0	Mov_SS_1	401003	401130	false		?
1	Mov_SS_1	401003	401128	false		?
2	Mov_SS_1	401003	401115	false		?
3	Mov_SS_1	401003	401102	false		?
4	Mov_SS_1	401003	4010ef	false		?
5	Mov_SS_1	401003	401068	true	yes	true
6	Mov_SS_1	401003	40105f	true	no	?

Figure 4.11: Created frames during analysis of PEB!IsDebugger technique obfuscated by code transposition.

4.3.6.2 Source/Sink Identification

With the same rule as before, the rule engine was able to find a pair of instructions that began with a source node writing to the `SS` register and terminating at a conditional jump sink node. The frame at index five begins at line `0x401003` and ends at `0x401068`; including the opaque predicate code, this is correct for the MOV SS technique. With one exception, the other frames were discarded after failing the chop attempt. Index six identified the dummy code for the opaque predicate.

4.3.6.3 Chop Identification

REDIR attempted to chop the program for each of the six frames generated by the rule engine. Index five, beginning with the source and sink nodes `0x401005` and `0x401014`, managed the correct source and sink nodes necessary for *iltrans* to create a chop. Additionally, while REDIR was not designed to identify specific obfuscations it did provide an important clue to the use of an opaque predicate. Closer inspection of the two frames shows frame five is the same as frame six with two additional instructions. Index

```
00401000    xor ebx, ebx
00401002    push
00401003    pop
00401004    pushf
00401005    xor eax, eax
00401007    add ax, word_403026
0040100E    add ax, word_403028
00401015    imul ax, ax
00401019    push eax
0040101B    xor eax, eax
0040101D    mov ax, word_403026
00401023    imul ax, ax
00401027    push eax
00401029    xor eax, eax
0040102B    mov ax, word_403028
00401031    imul ax, ax
00401035    push eax
00401037    xor eax, eax
00401039    xor ebx, ebx
0040103B    mov ax, word_403026
00401041    mov bx, word_403028
00401048    imul ax, bx
0040104C    imul ax, 2
00401050    pop ebx
00401052    add ax, bx
00401055    pop ebx
00401057    add ax, bx
0040105A    pop ebx
0040105C    cmp ax, bx
0040105F    jnz short loc_401076
00401061    test small word ptr [esp+1], 1
00401068    jnz short loc_401086
0040106A    push 40300Fh
0040106F    call sub_401098
00401074    jmp short _start
```

Figure 4.12: Highlighted DigR disassembly view of PEB!IsDebugger technique obfuscated by code transposition.

six did lead to a valid chop, however the technique was not detectable. During debugging, an analyst could use this clue to explore and confirm the presence of an opaque predicate.

4.3.6.4 Anti-debugging Technique Identification

Again, just one MOV SS implementation was present in the program. Frame six was eventually discarded when the evaluation could not detect the technique. The frame at index five correctly identifies that implementation.

4.3.6.5 Jump Direction

By simulating setting the TF, REDIR correctly identified that using a debugger would cause this program to take the jump (to line `0x401086`) rather than falling through to the next instruction (at line `0x40106a`).

4.4 Design and Implementation Analysis

REDIR excelled at many of the stated goals. The method for capturing detections by the Data/Frame sensemaking technique seems to be a valid starting point for future research. REDIR created and evaluated frames for correctness before employing more demanding analysis steps. This approach greatly reduced the problem search space and minimized expensive analysis steps by concrete evaluation.

REDIR offered additional benefits that were not originally intended. The original design for confirming the presence of an anti-debugging technique also offered consistent detection of a technique's designed jump direction. Additionally, as evident in Test Case #6 (Section 4.3.6), REDIR demonstrated value by offering a clue for an analyst to explore to confirm an obfuscation.

Most REDIR analysis tasks completed in less than one second. However, due to the less-than optimal multiple Virtual Machine (VM) architecture, execution time results could not be viewed as meaningful metrics.

4.5 Experiment Summary

This research selected a test methodology to demonstrate the feasibility of static analysis by sensemaking and IR analysis. In that task, the REDIR system was very

successful. However, many other anti-debugging and obfuscation techniques exist. An exhaustive test of all known techniques was beyond the scope of this project. Definitive tests for real-world malware samples were impossible with static-only analysis, therefore REDIR did not test real-world samples.

V. Conclusion

5.1 Overview

The purpose of this chapter is to summarize the research conducted for this thesis. Section 5.2 discusses the significance of this research. Section 5.3 offers new paths to progress this research forward. Finally, Section 5.4 serves to summarize this entire research effort.

5.2 Research Significance

As described previously, Reverse Code Engineering (RCE) is a time-consuming and complicated task that requires a high level of education and expertise. Tools to help RCE analysts conduct their work can make these analysts more effective in their work. Complicating the work of the RCE analyst, anti-debugging techniques compound the difficulty of RCE. Tools exist to detect anti-debugging code but they are susceptible to obfuscations. The purpose of REDIR is to detect anti-debugging techniques in obfuscated code.

The REDIR system has managed to achieve its intended purpose. REDIR has successfully detected three different anti-debugging techniques in six different obfuscations. This success demonstrates the feasibility of the system and encourages continued development.

Furthermore, REDIR has demonstrated the effectiveness of three different concepts for RCE tasks. First, the Data/Frame sensemaking theory was justified as an effective method for growing possible detections into confirmed detections. Next, the use of an Expert System (ES), particularly a rule-based ES, made simple work of finding the minimal heuristics of a technique for further processing. Finally, the Intermediate Representation (IR) technology provided by the Binary Analysis Platform (BAP) framework successfully

revealed the test programs inner workings and provided concrete evaluation that ultimately made the detection confirmations possible.

5.3 Future Research Recommendations

5.3.1 REDIR Enhancements

The REDIR system was designed to demonstrate the feasibility of a static, anti-debugging detection system based on IR. As a result, the system is only partially implemented. Several enhancements can be made to REDIR that will enable more capabilities and approach commercial capabilities. Many others exist. REDIR should be extended to detect more techniques. Some techniques were not possible due to limitations imposed by the BAP framework. If BAP develops to handle cycles, REDIR can detect techniques comprised of loops. When BAP can deploy to a Windows environment, REDIR should be redesigned to eliminate the multiple Virtual Machine (VM) architecture.

REDIR's rule engine implementation allows for the addition and removal of data in working memory. Extending REDIR into dynamic analysis will mitigate the issue with cycles. The DigR debugger can provide dynamic trace data while debugging. At each breakpoint or single-step in the debugger, replacing the static IR with dynamic trace data converted to IR will enable the re-firing of rule engine rules with the possibility of new detections. Advancing over cycles will replace loops with the sequence of executed instructions. This enhancement will allow for detection of anti-debugging techniques that form cycles, mitigation of obfuscations that employ cycles, and detection of decrypted or decompressed of anti-debugging techniques. Furthermore, this modification is supported by the Data/Frame sensemaking process. The *reframing* step encourages the creation of new frames by seeking additional data. Figure 5.1 depicts the REDIR concept through the Data/Frame sensemaking process with the addition of dynamic trace data.

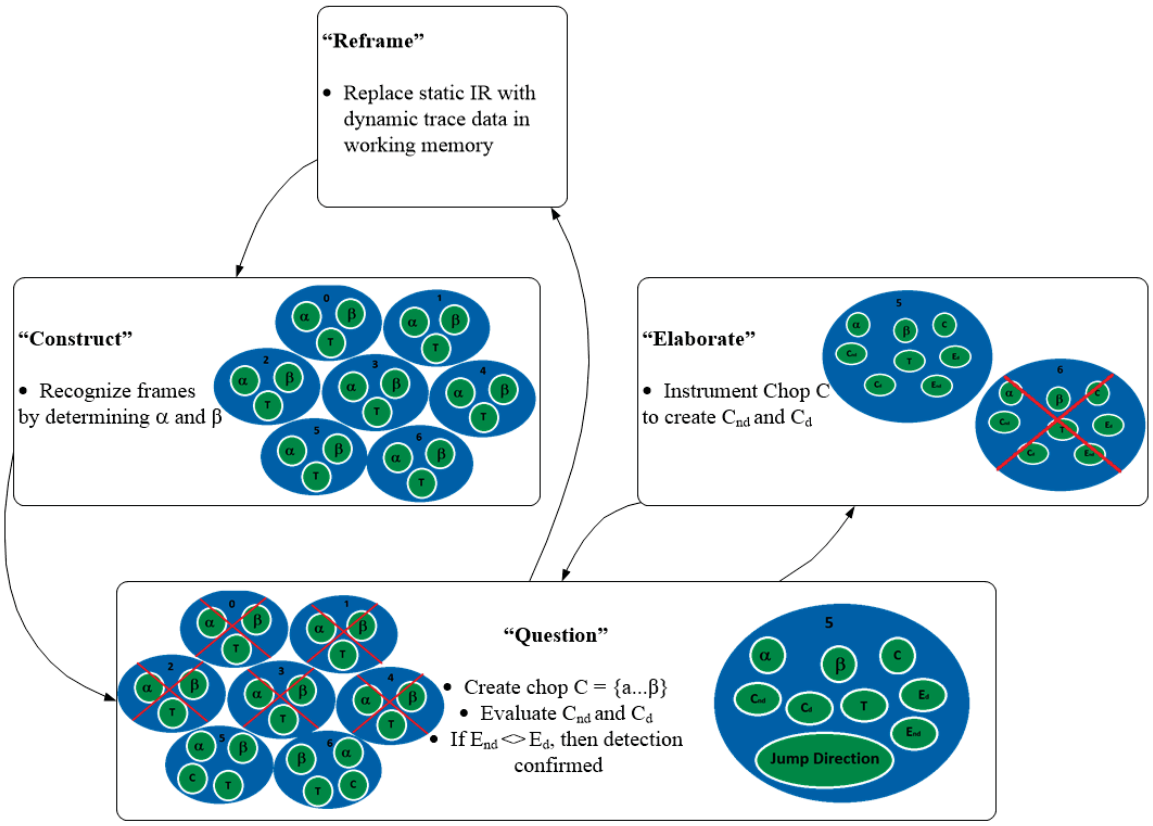

Figure 5.1: REDIR concept through the Data/Frame sensemaking process with additional dynamic trace data.

5.3.2 Test Corpus Development

During pilot experimentation, this research considered many other anti-debugging and obfuscation technologies. However, a complete test corpus of all known technologies was beyond the scope of this research. A complete test corpus would enable future researchers to delay testing on real-world malware samples and focus on simple, provable analyses. Additionally, other transformations such as encryption, packing, and integration into existing programs could extend this test corpus and present a close to real-world malware simulation. Lastly, working with real-world malware requires special handling to prevent accidental execution on non-testing platforms. Extending this test corpus would

provide a known benign dataset. Test cases could be used during application development without fear of malware infection.

5.3.3 Applications for Expert Systems Technologies in Reverse Code Engineering Tasks

During the initial phases of this research effort, numerous different pairings of ES and RCE were considered. The following subsections enumerate these research paths to describe possible applications of ES in RCE based on strengths, weaknesses and characteristics of each. This is not an exhaustive list of possible applications. Many of the ES concepts can apply in various hybrid forms, and any of the issues around RCE can merge into a single solution.

5.3.3.1 Ontology-Based Unpacker Tool

The use of packers is very common in malware development. Packers make analysis very difficult and consume a large portion of the reverse engineer's time. Current techniques for unpacking malware consist of manual debugging and automated scripts. Debugging is time consuming and scripts are only useful for particular packers. Each time a new packer is encountered, debugging and script development start over. Malware analysts need a tool that can automate the debugging of any packed malware, including never before encountered packers. Perhaps an ES-based unpacker could fill this need.

An implementation for an ES-based unpacker would need to do several things. First, it must formalize the knowledge of Subject Matter Experts (SMEs). Any ES technology could accomplish this, but this requirement could be most easily and explicitly attained in a rule-based or ontology-based knowledge. Once created, this data can be added to the reasoner attached to a debugger program. The system must be capable of debugging programs in execution to apply the knowledge as necessary to determine the Original Entry Point (OEP) for the program. Each program the system will analyze will be different; as

a result, the use of the knowledge cannot be rigid. For this reason, an ontology-based tool would be the best choice.

To mitigate the cost of creating the ontology, a complete ontology of all known unpacking strategies and anti-reverse engineering techniques should not be attempted at first. Since different packer implementations employ many of the same anti-reverse engineering techniques, rules pertaining to unpacking should be added one packer at a time. This strategy will be useful to cover the most commonly used anti-reverse engineering first. Subsequent additions to the ontology will require fewer additions to achieve the same result. Future packers may require no additions at all as their anti-reverse engineering strategies previously added to the ontology.

5.3.3.2 Intelligent Tutoring System For Teaching Reverse Code Engineering Concepts

There are several common areas of knowledge deficiency for new RCE students [63]. They must learn assembly programming, compiler optimization behavior and other specialized skills that typical Computer Science curriculum do not provide. The availability of an Intelligent Tutoring System (ITS) in those areas would offer the RCE student self-paced, goal-oriented instruction. With the prerequisite instruction completed, the student could begin RCE education. Additionally, ITS systems could aid the engineer in the difficult cognitive processes present in most reverse engineering tasks.

5.3.3.3 Modeling Domain Explicit Knowledge with Rule-Based Expert System

If-Then-Else logic used as a system's knowledge base characterizes rule-based systems. Simple, explicit RCE rules are programmable into the knowledge base for a rule-based system. For example: a simple RCE task could be represented as a rule: `ruleid: If section_encrypted then attempt_decryption`. Based on previous work done with

rule-based systems such as the LISP tutor, a similar approach could be applied in a RCE tutoring system [17].

5.3.3.4 Capturing Subject Matter Expert Knowledge with a Knowledge-Based Expert System

Knowledge-based ES depend on quality domain knowledge from experienced sources. Capturing this knowledge is the hardest part of building a knowledge-based system. The knowledge of RCE experts has not been encapsulated in any knowledge base. A knowledge-based tool with an integrated RCE tool interface could collect and learn expert knowledge for later use.

5.3.3.5 An Ontology-Based Reverse Code Engineering Sensemaking System

Currently, there is not an ontology pertaining specifically to RCE. An ontology that encapsulated and categorized the tools, techniques and foundational knowledge would provide domain knowledge in an electronic form for use directly, or extracted from, to create other RCE applications. Reverse engineers need better tools for documenting their progress and sharing information [70]. An ontology-based documentation/collaboration system could provide sensemaking assistance to help reverse engineers document solution paths as they build a representative model of the system to share. Additionally, the ontology could bridge knowledge and experience between reverse engineers and a sensemaking system. The ontology would lay the foundation for the predictability between task workers and the system required for an effective sensemaking system.

5.3.3.6 Fuzzy Logic in a Knowledge Base Query Application

Usability is a critical design feature in all software applications. A knowledge base is no good if the reverse engineer cannot construct queries that will provide the information they are looking for. A query engine that will forgive spelling mistakes and offer results based on synonyms could provide best match results based on a human user's input.

5.3.3.7 Feature Recognition with Case-Based Reasoning

A Case-Based Reasoning (CBR) exploratory learning environment would be useful for RCE feature recognition activities. Reverse engineers would be able to identify parts of a program as new features. Once verified, those features could be added to the Database (DB) of past cases of that particular feature. Future RCE applications could use that DB as the foundation of automated feature detection.

5.3.3.8 De-obfuscation via Hidden Markov Models

Due to obfuscations and anti-reverse engineering techniques, frivolous instructions intended to confuse the reverse engineer may be disguise common features of a program. A system employing Hidden Markov Models (HMMs) could predict the presence of features based on only partial sets of observations. Confirmed instances could add to a database of de-obfuscated "fingerprints" for future use.

5.4 Summary

In conclusion, this research has covered numerous topics in an attempt to address the problem of obfuscated anti-debugging techniques. The background and fundamentals necessary to understand the problem and its possible solution were described in Chapter 2. Chapter 3 presented a solution to the problem and detailed its design, implementation and method of testing. Subsequently, in Chapter 4, the experimentation results were presented accompanied by analysis of the implementation and experimental method.

The problems facing those who perform RCE are not getting easier. To the contrary, the domain is growing in complexity. The tools that reverse engineers depend on are not keeping up with this trend. It is important to look at new methods for improving the way that RCE is conducted. This research attempted to use a sensemaking strategy, driven by a rule-based ES, employing IR analysis to do just that. Hopefully, this research will lead to new RCE tools that incorporate proven technologies, such as ES and IR, to extend their functionality and improve the performance of the reverse engineers that use them.

Bibliography

[1] "Anti-Anti-Debugger Plugins," (n.d.). March 2014. [Online]. Available: https://code.google.com/p/aadp/

[2] "Anti-Debug Time Plugin for OllyDbg," July 2013. March 2014. [Online]. Available: http://www.codeproject.com/Articles/614775/Anti-Debug-Time-Plugin-for-OllyDbg

[3] "Cygwin," (n.d.). October 2013. [Online]. Available: http://www.cygwin.com/

[4] "DigR: An Integrated and Easy-to-use Tool Suite for Reverse Engineering," (n.d.). October 2013. [Online]. Available: http://www.riversideresearch.org/ird/digr-integrated-and-easy-use-tool-suite-reverse-engineering

[5] "Drools - The Business Logic Integration Platform," (n.d.). June 2013. [Online]. Available: https://www.jboss.org/drools/

[6] "Hermit OWL Reasoner," (n.d.). March 2013. [Online]. Available: http://www.hermit-reasoner.com/

[7] "Intel 64 and IA-32 Architectures Software Developers Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C," Feburary 2014. Feburary 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[8] "MinGW - Minimalist GNU for Windows," (n.d.). October 2013. [Online]. Available: http://www.mingw.org/

[9] "OWL Web Ontology Language Overview," November 2009. March 2013. [Online]. Available: http://www.w3.org/TR/owl-features/

[10] "Pellet: OWL 2 Reasoner for Java," (n.d.). March 2013. [Online]. Available: http://clarkparsia.com/pellet/

[11] "Tuts4You," (n.d.). March 2014. [Online]. Available: https://tuts4you.com/download.php

[12] A. Aamodt and E. Plaza, "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches," *AI Communications*, vol. 7, no. 1, pp. 39–59, 1994.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2007, vol. 1009.

[14] R. Akerkar and P. Sajja, *Knowledge-Based Systems*. Jones & Bartlett Learning, 2010.

[15] E. Alpaydn, *Introduction to Machine Learning*, 2nd ed. Massachusetts Institute of Technology, 2010.

[16] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier, "Cognitive Tutors: Lessons Learned," *The Journal of the Learning Sciences*, vol. 4, no. 2, pp. 167–207, 1995.

[17] J. Anderson and B. Reiser, "The LISP tutor," *Byte*, vol. 10, no. 4, pp. 159–175, 1985.

[18] G. Arboit, "A Method for Watermarking Java Programs via Opaque Predicates," in *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[19] G. Balakrishnan and T. Reps, "WYSINWYX: What you see is not what you eXecute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, p. 23, 2010.

[20] I. Becerra-Fernandez, "Technologies to Manage Knowledge: Artificial Intelligence," 2004. [Online]. Available: http://www.cse.ust.hk/~dekai/600G/notes/KM_Slides_Ch07.pdf

[21] A. Ben-David and E. Frank, "Accuracy of Machine Learning Models Versus "Hand Crafted" Expert Systems–A Credit Scoring Case Study," *Expert Systems with Applications*, vol. 36, no. 3, Part 1, pp. 5264–5271, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417408003904

[22] B. Bloom, "The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring," *Educational Researcher*, vol. 13, no. 6, pp. 4–16, 1984.

[23] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Computer Aided Verification*. Springer, 2011, pp. 463–469.

[24] D. Brumley, I. Jager, E. J. Schwartz, and S. Whitman, "The BAP Handbook," 2013. August 2013.

[25] A. Bryant, R. Mills, G. Peterson, and M. Grimaila, "Software Reverse Engineering as a Sensemaking Task," *Journal of Information Assurance and Security*, vol. 6, no. 6, pp. 483–494, 2012.

[26] A. I. Center and J. Semarak, "Intelligent Traffic Lights Control by Fuzzy Logic," *Malaysian Journal of Computer Science*, vol. 9, no. 2, pp. 29–35, 1996.

[27] H. Cha, Y. Kim, S. Park, T. Yoon, Y. Jung, and J.-H. Lee, "Learning Styles Diagnosis Based on User Interface Behaviors for the Customization of Learning Interfaces in an Intelligent Tutoring System," in *Intelligent Tutoring Systems*. Springer, 2006, pp. 513–524.

[28] Y. F. Chan, H. K. Ma, F. T. Chan, H. Y. Chen, and T. Y. Chen, "Teaching Family Planning with Expert System," *Comput. Educ.*, vol. 24, no. 4, pp. 293–298, May 1995.

[29] C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1998, pp. 184–196.

[30] A. T. Corbett, K. R. Koedinger, and J. R. Anderson, "Intelligent Tutoring Systems," *Handbook of Human-Computer Interaction*, pp. 849–874, 1997.

[31] J. Durkin, "Research Review: Application of Expert Systems in the Sciences," *The Ohio Journal of Science*, vol. 90, no. 5, pp. 171–179, 1990.

[32] S. Dutta, "Strategies for Implementing Knowledge-Based Systems," *Engineering Management, IEEE Transactions on*, vol. 44, no. 1, pp. 79–90, 1997.

[33] S. R. Eddy, "Profile Hidden Markov Models," *Bioinformatics*, vol. 14, no. 9, pp. 755–763, 1998.

[34] E. Eilam, *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, 2005.

[35] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis," in *Software Maintenance, 2001. Proceedings. IEEE International Conference on*. IEEE, 2001, pp. 602–611.

[36] K. A. Ericsson, M. J. Prietula, and E. T. Cokely, "The Making of an Expert," *Harv. Bus. Rev.*, vol. 85, no. 7/8, p. 114, 2007.

[37] J. Ferguson, D. Kaminsky, J. Larsen, L. Miras, and W. Pearce, *Reverse Engineering Code with IDA Pro*. Syngress Publishing, 2008.

[38] P. Ferrie, "The "Ultimate" Anti-Debugging Reference," 2011. March 2014. [Online]. Available: http://pferrie.host22.com/papers/antidebug.pdf

[39] N. Gardan and Y. Gardan, "An Application of Knowledge-Based Modelling Using Scripts," *Expert Systems with Applications*, vol. 25, no. 4, pp. 555–568, 2003.

[40] S. Gutierrez-Santos, M. Cocea, and G. Magoulas, "A Case-Based Reasoning Approach to Provide Adaptive Feedback in Microworlds," in *Intelligent Tutoring Systems*. Springer, 2010, pp. 330–333.

[41] I. Hatzilygeroudis and J. Prentzas, "Integrating (Rules, Neural Networks) and Cases for Knowledge Representation and Reasoning in Expert Systems," *Expert Systems with Applications*, vol. 27, no. 1, pp. 63–75, 2004.

[42] ——, "Using a Hybrid Rule-Based Approach in Developing an Intelligent Tutoring System with Knowledge Acquisition and Update Capabilities," *Expert Systems with Applications*, vol. 26, no. 4, pp. 477–492, 2004.

[43] A. He, K. K. Bae, T. Newman, J. Gaeddert, K. Kim, R. Menon, L. Morales-Tirado, J. Neel, Y. Zhao, J. Reed, and W. Tranter, "A Survey of Artificial Intelligence for Cognitive Radios," *Vehicular Technology, IEEE Transactions on*, vol. 59, no. 4, pp. 1578–1592, May 2010.

[44] K. Higa and H. G. Lee, "A Graph-Based Approach for Rule Integrity and Maintainability in Expert System Maintenance," *Information & Management*, vol. 33, no. 6, pp. 273–285, Jun. 1998.

[45] A. Honig, *Practical Malware Analysis*. No Starch Press, 2012.

[46] C.-W. Hsu, S. W. Shieh *et al.*, "Divergence Detector: A Fine-Grained Approach to Detecting VM-Awareness Malware," in *Software Security and Reliability (SERE), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 80–89.

[47] K. R. Irvine, *Assembly Language for x86 Processors*. Prentice Hall, 2011.

[48] J. H. Jahnke and J. Wadsack, "The Varlet Analyst: Employing Imperfect Knowledge in Database Reverse Engineering Tools," in *in Database Reverse Engineering Tools. 3rd International Workshop on Intelligent Software Engineering (WISE-3*, 2000, pp. 59–69.

[49] J. H. Jahnke and A. Walenstein, "Reverse Engineering Tools as Media for Imperfect Knowledge," in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. IEEE, 2000, pp. 22–31.

[50] G. Klein, B. Moon, and R. Hoffman, "Making Sense of Sensemaking 1: Alternative Perspectives," *Intelligent Systems, IEEE*, vol. 21, no. 4, pp. 70–73, 2006.

[51] ——, "Making Sense of Sensemaking 2: A Macrocognitive Model," *Intelligent Systems, IEEE*, vol. 21, no. 5, pp. 88–92, 2006.

[52] G. Klein, D. D. Woods, J. M. Bradshaw, R. R. Hoffman, and P. J. Feltovich, "Ten Challenges for Making Automation a "Team Player" in Joint Human-Agent Activity," *Intelligent Systems, IEEE*, vol. 19, no. 6, pp. 91–95, 2004.

[53] R. Koschke, J.-F. Girard, and M. Wurthner, "An Intermediate Representation for Integrating Reverse Engineering Analyses," in *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*. IEEE, 1998, pp. 241–250.

[54] L.-F. Lai, C.-C. Wu, P.-Y. Lin, and L.-T. Huang, "Developing a Fuzzy Search Engine Based on Fuzzy Ontology and Semantic Search," in *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2684–2689.

[55] S.-H. Liao, "Expert System Methodologies and Applications–A Decade Review from 1995 to 2004," *Expert Systems with Applications*, vol. 28, no. 1, pp. 93–103, 2005.

[56] A. Majumdar, C. Thomborson, and S. Drape, "A Survey of Control-Flow Obfuscations," in *Information Systems Security*. Springer, 2006, pp. 353–356.

[57] R. S. Mitra and A. Basu, "Knowledge Representation in MICKEY: An Expert System for Designing Microprocessor-Based Systems," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 27, no. 4, pp. 467–479, 1997.

[58] A. Mitrovic, "An Intelligent SQL Tutor on the Web," *International Journal of Artificial Intelligence in Education*, vol. 13, no. 2-4, pp. 173–197, 2003.

[59] R. J. Mockler, D. G. Dologite, and M. E. Gartenfeld, ""Talk with the Experts": Learning Management Decision-Making Using CAI," *Cybernetics & Systems*, vol. 31, no. 4, pp. 431–464, 2000.

[60] H. Nwana, "Intelligent Tutoring Systems: an Overview," *Artificial Intelligence Review*, vol. 4, no. 4, pp. 251–277, 1990.

[61] G. Paviotti, P. Rossi, and D. Zarka, *Intelligent Tutoring Systems: an Overview*. Pensa Multimedia, 2012.

[62] M. Pietrek, "Inside Windows-An In-Depth Look into the Win32 Portable Executable File Format, Part 2," *MSDN magazine*, pp. 87–100, 2002.

[63] G. Richard III, "A Highly Immersive Approach to Teaching Reverse Engineering," in *Proceedings of the 2nd Workshop on Cyber Security Experimentation and Test (CSET 2009)*, 2009.

[64] J. M. Ruiz-Sánchez, R. Valencia-Garcıa, J. T. Fernández-Breis, R. Martınez-Béjar, and P. Compton, "An Approach for Incremental Knowledge Acquisition from Text," *Expert Systems with Applications*, vol. 25, no. 1, pp. 77–86, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417403000083

[65] M. Sasikumar, S. Ramani, S. M. Raman, K. Anjaneyulu, and R. Chandrasekar, *A Practical Introduction to Rule Based Expert Systems*. Narosa Publishing House, 2007.

[66] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding Malware Analysis Using Conditional Code Obfuscation," in *NDSS*, 2008.

[67] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[68] Y. Takaoka and R. Mizoguchi, "Identification of Ontologies to Reuse Knowledge for Substation Fault Recovery Support System," *Decision Support Systems*, vol. 18, no. 1, pp. 3–21, 1996.

[69] S. Treadwell and M. Zhou, "A Heuristic Approach for Detection of Obfuscated Malware," in *Intelligence and Security Informatics, 2009. ISI'09. IEEE International Conference on*. IEEE, 2009, pp. 291–299.

[70] C. Treude, F. Filho, M. Storey, and M. Salois, "An Exploratory Study of Software Reverse Engineering in a Security Context," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, Oct. 2011, pp. 184–188.

[71] M. Uschold, M. Gruninger *et al.*, "Ontologies: Principles, Methods and Applications," *Knowledge Engineering Review*, vol. 11, no. 2, pp. 93–136, 1996.

[72] K. Vanlehn, "The Behavior of Tutoring Systems," *International Journal of Artificial Intelligence in Education*, vol. 16, no. 3, pp. 227–265, 2006.

[73] A. Walenstein, "Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework," Ph.D. dissertation, Simon Fraser University, 2002.

[74] T.-P. Wu and S.-M. Chen, "A New Method for Constructing Membership Functions and Fuzzy Rules from Training Examples," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 29, no. 1, pp. 25–40, Feb. 1999.

[75] P. Xie, X. Lu, Y. Wang, J. Su, and M. Li, "An Automatic Approach to Detect Anti-Debugging in Malware Analysis," in *Trustworthy Computing and Services*. Springer, 2013, pp. 436–442.

[76] I. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE, 2010, pp. 297–300.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2014	Master's Thesis	Oct 2013–Mar 2014

4. TITLE AND SUBTITLE

REDIR: Automated Static Detection of Obfuscated Anti-Debugging Techniques

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Smith, Adam J., Technical Sergeant, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-14-M-69

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Riverside Research, Cyber Research Laboratory
Attn: Adam R. Bryant
2640 Hibiscus Way
Beavercreek, OH 45431
abryant@RiversideResearch.org

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Reverse Code Engineering (RCE) to detect anti-debugging techniques in software is a very difficult task. Code obfuscation is an anti-debugging technique makes detection even more challenging. The ***Rule Engine Detection by Intermediate Representation*** (REDIR) system for automated static detection of obfuscated anti-debugging techniques is a prototype designed to help the RCE analyst improve performance through this tedious task. Three tenets form the REDIR foundation. First, Intermediate Representation (IR) improves the analyzability of binary programs by reducing a large instruction set down to a handful of semantically equivalent statements. Next, an Expert System (ES) rule-engine searches the IR and initiates a sensemaking process for anti-debugging technique detection. Finally, an IR analysis process confirms the presence of an anti-debug technique. The REDIR system is implemented as a debugger plug-in. Within the debugger, REDIR interacts with a program in the disassembly view. Debugger users can instantly highlight anti-debugging techniques and determine if the presence of a debugger will cause a program to take a conditional jump or fall through to the next instruction.

15. SUBJECT TERMS

Reverse code engineering, Expert systems, Sensemaking, Anti-debugging

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Robert F. Mills (ENG)
U	U	U	UU	99	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4527 robert.mills@afit.edu